А. Ф. Губкин

ОПЕРАЦИОННЫЕ СИСТЕМЫ

2019

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Идея настоящей работы состоит в том, чтобы развеять у читателя представление о том, что в компьютере сидит некоторый маг или еще кто-нибудь всесильный, который всем управляет и все может. Такой взгляд встречается и в некоторых изданиях. Особенно поражает использование терминов вытесняющей и невытесняющей многозадачности, когда более приоритетный процесс вытесняет с центрального процессора менее приоритетный процесс. Реально замену процессов осуществляет функция управляющей программы Диспетчер. Эта функция получает управление либо по окончанию кванта времени, который был предоставлен процессу Диспетчером перед его стартом, либо Диспетчер получает управление после завершения работы текущего процесса на центральном процессоре.

Собственно описанию и объяснению работы таких механизмов и посвящена эта книга. Конечно эти механизмы и алгоритмы различаются в разных типах операционных систем (ОС). Поэтому она не посвящена описанию какой-то одной ОС или семейству родственных систем, а излагает общие вопросы, связанные с проблематикой ОС. К числу таких вопросов относятся определение понятия ОС, выделение типов ОС, командный интерфейс с пользователем и программный интерфейс приложений, типы и организация исполняемых программных модулей, обсуждение методов и алгоритмов управления основной памятью, управления процессами и ресурсами, управление устройствами, вводом/выводом, данными.

Знаниями по этим вопросам должны обладать все квалифицированные специалисты в области информационных технологий (ИТ), поскольку в настоящее время существует большое количество различных аппаратно-программных платформ самого разнообразного назначения. Это и клиентские станции, и серверное обеспечение, и обеспечение мэйнфреймов, не говоря о мобильных устройствах.. Эти знания помогут быстрее и глубже освоить конкретную ОС в среде, которой придется работать специалисту либо занимаясь эксплуатацией в качестве системного администратора или администратора базы данных, либо в качестве разработчика программного обеспечения (ПО).

Разумеется, предлагаемый курс является лишь составной частью множества дисциплин, необходимых для получения качественного образования в области ИТ и предполагает, что предварительно были получены знания и практические навыки по следующим дисциплинам. Прежде всего, по дисциплине «Программирование и конструирование программ». Необходимо, чтобы был опыт разработки программ на ассемблере и языке высокого уровня в

какой-либо среде программирования (например, C/C++). Дисциплина «Структуры данных» также необходима, хотя бы потому, что структуры данных это фольклор программиста. Необходимы знания и практические навыки в области конструирования компиляторов. Также необходимы знания по аппаратному обеспечению в объеме курса «Архитектура и организация вычислительных систем».

Предлагаемый курс ОС состоит из двух частей: «Введение в ОС» и «Организация и функционирование ОС». Соотношение этих частей иллюстрирует Рис. 1.

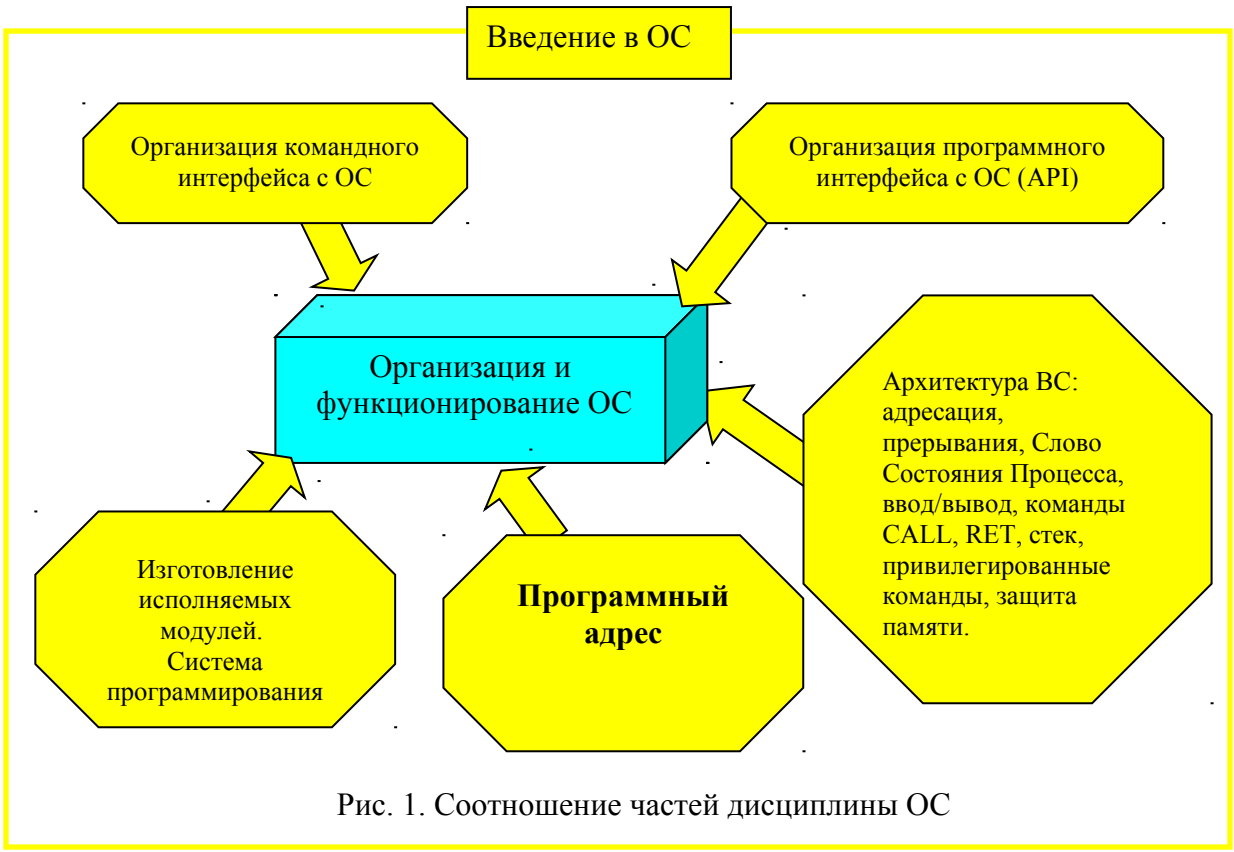


Рис. 1. Соотношение частей дисциплины ОС

В части «Введение в ОС» рассматриваются общие вопросы, необходимые для понимания организации и функционирования ОС. Во второй части рассматривается управляющая программа, то есть та часть, которая обеспечивает управление основной памятью (ОП), процессами, внешними устройствами, вводом/выводом, данными.

Опыт работы с командным интерфейсом имеют все пользователи. По крайней мере с графическим интерфейсом. Менее известно об организации программного интерфейса.

Важно знать организацию и архитектуру компьютерных систем. Это организация и функционирование системы прерываний, маскирование прерываний, система защиты памяти, состояния процессора и флаги, характеризующие эти состояния. Необходимо также иметь представление о привилегиро-

ванных командах процессора и их использовании. Также необходимо знать различные типы адресации, используемые в процессорах, команды работы со стеком и команды, поддерживающие вызовы подпрограмм. Эти сведения используются, но не рассматриваются подробно в этом курсе.

В книге приводятся примеры из используемых в настоящее время ОС. Это ОС фирмы IBM для мэнфрэймов z/OS MVS и диалоговых систем UNIX и Linux. Следует заметить, что z/OS несравненно сложнее и поэтому примеры из этой системы занимают большее место в изложении. Хотя управление данными интереснее рассматривать в диалоговых системах.

Часть I. ВВЕДЕНИЕ В ОС

В этой части рассматриваются вопросы, связанные с определением понятия ОС. Действительно, это понятие употребляется в различных смыслах. В популярных изданиях, в обиходе под ОС обычно понимают все программное обеспечение, присутствующее на компьютере. При покупке ОС или скачивании свободно распространяемого экземпляра пользователь сталкивается с дистрибутивом системы, который определяется конкретной версией и релизом системы и имеет конкретный состав функций и программных средств.

В специальной литературе, посвященной описанию устройства и организации ОС, рассматривается еще более узкий состав функций и программных средств, поддерживающий управление процессами и ресурсами ОС.

Также в этой части рассматриваются различные типы ОС, их принципиальные различия с точки зрения областей применения. Однако, выделение областей применения осуществляется с учетом влияния на структуру и алгоритмы управления, применяемые в различных ОС.

Основной задачей этой части является обсуждение вопросов изготовления загрузочных модулей, способов их загрузки в основную память и преобразования адресов в процессе подготовки к выполнению и в процессе выполнения. Также рассматриваются различные типы загрузочных модулей и способы их организации.

Тема 1. ОПРЕДЕЛЕНИЕ ПОНЯТИЯ ОС

1.1. Состав программного обеспечения вычислительной системы

Многие авторы отмечают, что трудно дать всеобъемлющее и точное определение понятия ОС. В существующей литературе сложилось два понимания этого термина. Для того чтобы разобраться в различных смыслах, который вкладывается в понятие ОС, следует классифицировать весь спектр программ, функционирующих в среде компьютерной системы (КС). На рис. 2 представлена такая классификация.

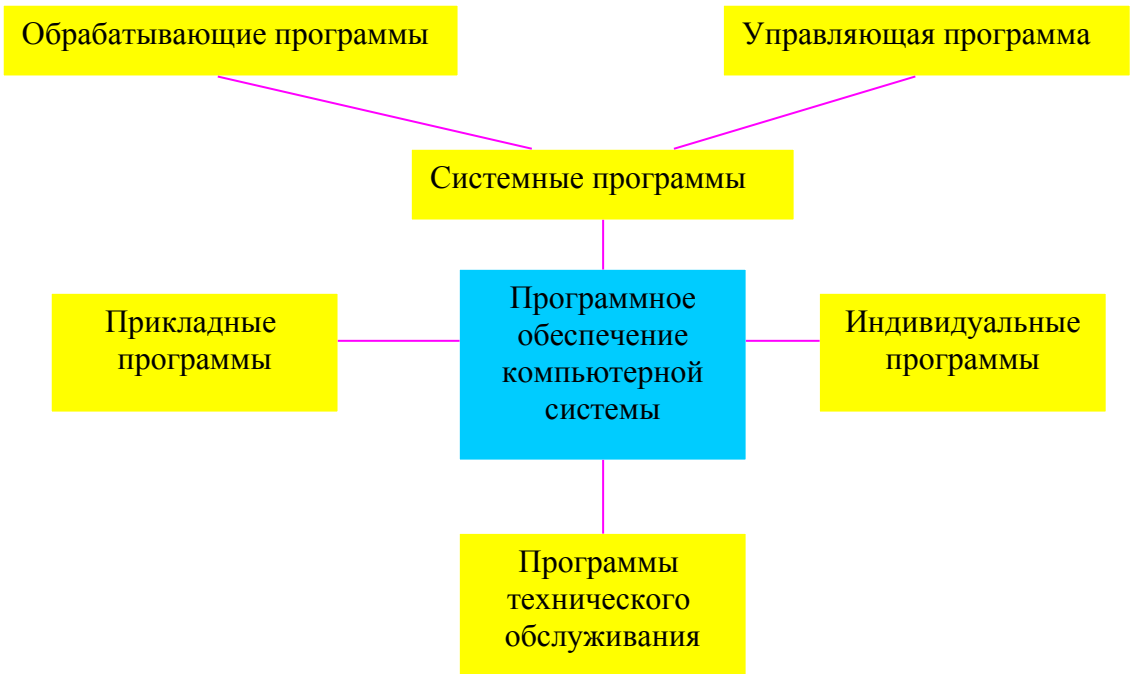


Рис. 2. Классификация программного обеспечения КС

Индивидуальные программы – это программы, которые пишут для личных целей. Это может быть проверка или иллюстрация какого-то метода, расчет каких-то значений и так далее. Основная цель – получение быстрого результата. Индивидуальные программы разрабатываются без использования технологического процесса разработки, как правило, сразу пишется код, который по мере проникновения в задачу корректируется. Отсутствует какая-либо документация. В лучшем случае создается текстовый файл ReadMe, в котором описывается обращение к программе. Исходный код комментируется в зависимости от вкусов автора. Этот тип программ не может существовать без участия разработчика.

Прикладные программы – это рыночный продукт, который производится и распространяется в соответствии с существующими законами и нормами. Как и всякий продукт, прикладная программа отчуждена от своего раз-

работчика, то есть пользователь может эксплуатировать эту программу без его участия. Для этого она должна быть хорошо документирована, проведены определенные испытания с целью определения ее свойств и осуществляется ее сопровождение. Все это требует соблюдения определенного технологического процесса разработки. Интересно заметить, что впервые продажа программы была осуществлена в 1980 году фирмой IBM. Этот год можно считать первым появлением рыночных отношений в сфере программного обеспечения. До этого ПО поставлялось совместно с компьютером и было привязано к этой модели, а порой и к поставляемому экземпляру.

Архитектура пакетов прикладных программ.

Прикладные программы могут иметь различную архитектуру. Так они могут поставляться в виде библиотеки объектных модулей или классов. Это могут быть библиотеки статистики или линейной алгебры. Это могут быть Java классы. Общим для такой архитектуры прикладных программ является то, что их пользователи должны быть достаточно квалифицированными программистами.

Другой вид пакетов прикладных программ с точки зрения их архитектуры представляется в виде системы программирования. Для какой-то объектной области строится простой язык, способный описывать объекты, отношения и действия в данной предметной области. Конечно, этот язык должен быть близок и понятен специалистам этой предметной области, а программистские способности пользователей могут быть уже существенно хуже, чем у пользователей библиотек. Для этого языка строится компилятор, который генерирует программу из библиотеки функций, поддерживающих данную предметную область. Такой способ организации использовался во времена, когда средства визуализации отсутствовали. С развитием средств отображения информации эта архитектура прикладных программ уступила место программным системам.

Архитектура программных систем предполагает визуальный графический и звуковой интерфейс, простой и понятный доступ к действиям в предметной области. Примеров прикладных программных систем великое множество. Для таких систем пользователю важнее иметь квалификацию в предметной области, а программистские навыки могут быть минимальны. Например, для того, чтобы редактировать фотографии с помощью PhotoShop гораздо важнее знать, как обрабатывать изображения.

Пакеты прикладных программ часто включаются в дистрибутив системы. Это Paint, Windows Media Player, калькулятор и другие.

Программы технического обслуживания–тесно связаны с аппаратурой. Они служат для управления и обслуживания аппаратной части ВС. Широко известны такие программы как fdisk, пакет для анализа и тестирования SiSoft Sandra, тест для проверки и настройки монитора Nokia Test. Программы технического обслуживания распространяются как прикладные программы, а также включаются в состав ОС. Это программа дефрагментации диска или реализация в Windows управления дисками вместо ранее применяемой fdisk.

Обрабатывающие программы – это программы, которые выполняют различные вспомогательные функции, связанные с ведением журналов событий, поддержкой различных протоколов и другие действия. В различных системах они называются по-разному. Так в Windows это службы. Они не имеют окон и при обычной работе не видны. Их можно обнаружить лишь с помощью специальных средств. Например, запустив Диспетчер задач. В системах UNIX это демоны.

Эти программы обязательно входят в состав дистрибутива ОС и стартуют при загрузке.

Управляющая программа – представляет собой набор функций и данных, которые находятся в дистрибутиве, при инсталляции загружаются в специальные файлы, а при запуске системы загружаются в определенную область основной памяти и находятся там резидентно. Управляющую программу часто называют ядром системы. Ядро ОС как правило реализует функции управления основной памятью, процессами, устройствами, вводом/выводом, файлами. Следует заметить, что в современных системах существует тенденция сократить размер ядра. Так в ОС QNX ядро реализует только функции управления основной памятью, сообщениями, сигналами и прокси. В этом случае резидентную часть ОС называют микроядром. Остальные функции реализуются в виде системных процессов - менеджеров ресурсов. Так существует Task manager, File manager и другие.

Таким образом, различают системы с монолитным ядром и микроядром.

Теперь можно вернуться к вопросу о двух пониманиях термина ОС. В общей литературе, посвященной ИТ, под ОС понимают ту совокупность ПО, которая поставляется в дистрибутивах. Это понятие ОС в широком смысле. В специальной литературе, посвященной проблематике ОС, рассматриваются функции, составляющие управляющую программу, то есть ОС рассматривается в более узком смысле. Это функции по управлению процессами и ресур-

сами. Такое понимание ОС в широком и узком смыслах следует учитывать при чтении литературы.

1.2. Определение ОС

Согласно ГОСТ 15971-90 «Системы обработки данных. Термины и определения.»: «ОС – система программ, предназначенная для обеспечения определенного уровня эффективности цифровой вычислительной системы за счет автоматизированного управления ее работой и предоставляемого пользователям набора услуг».

В этом определении существует несколько ключевых понятий. Рассмотрим их.

1) «уровень эффективности» - характеризует качество работы системы. Известны два основных показателя уровня эффективности: пропускная способность и время реакции системы на события.

Показатель «пропускная способность» используется для так называемых пакетных систем, которые предназначены для решения пакета задач. Например, это может быть обработка поступающих статистических данных, обработка изображений, поступающих с космических аппаратов. Пропускная способность измеряется в количестве решенных задач в единицу времени.

Показатель «время реакции на события» характерен для систем, которые обрабатывают поток событий и на каждое событие должны выдавать определенную реакцию. Это может быть система заказа и покупки билетов (железнодорожных или авиа), система управления ядерным реактором и огромное количество других систем.

2) «автоматизированное управление работой» вычислительной системы состоит в управлении прохождением программ и управлении ее ресурсами. Этим занимается управляющая программа.

3) «набор услуг, предоставляемый пользователям» определяется назначением и версией ОС. Эта функция возложена на прикладные программы.

Рассмотренное выше определение говорит об ОС в широком смысле. Наиболее полно удовлетворяет этому определению дистрибутивы операционных систем.

1.3. Система программирования

ОС управляет выполнением программ, которые, собственно, и ведут необходимую пользователю обработку информации. Для того чтобы ОС могла это делать, необходимо представить программы в определенном стандартном формате – в виде загрузочных (или исполнимых) модулей. Эти модули

хранятся в файлах специального вида. Например, в Windows эти файлы имеют расширение .exe, а в MS DOS .exe. или com.

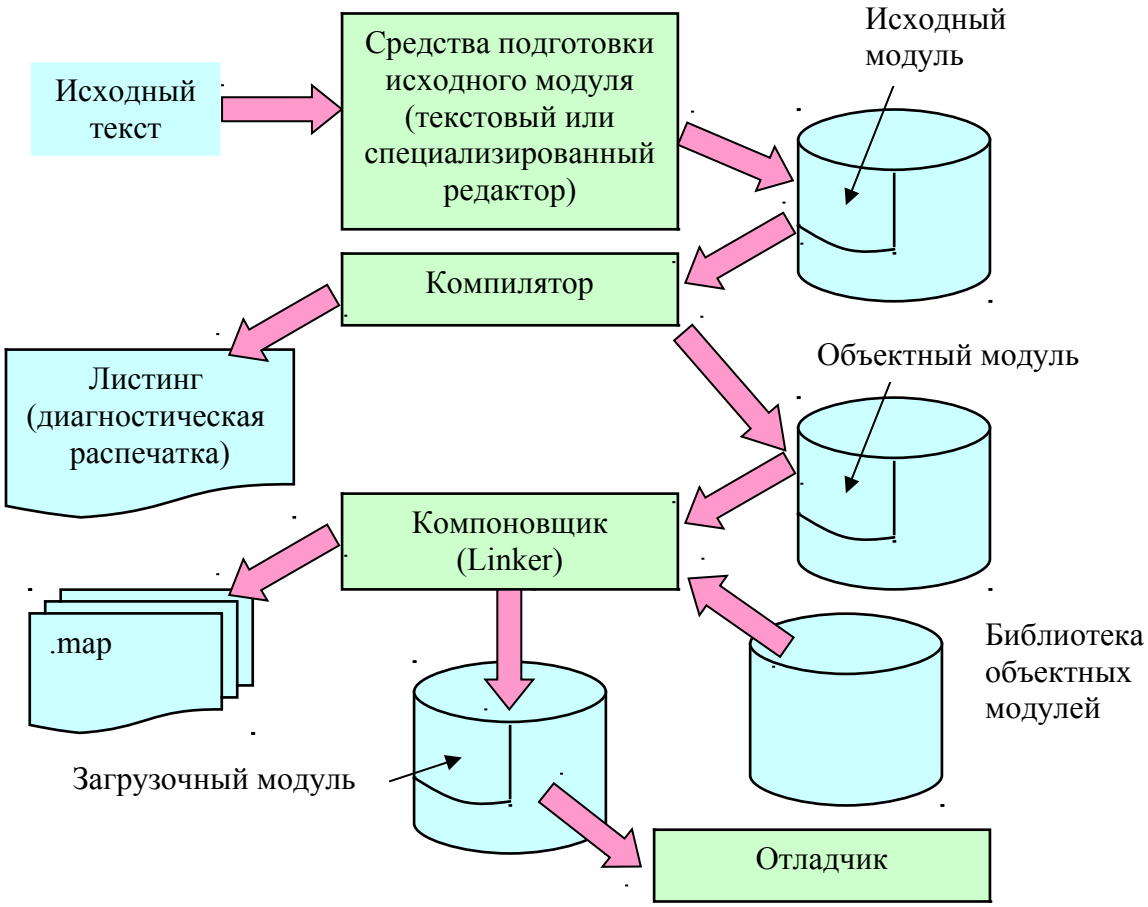


Рис. 3. Традиционная схема системы программирования

Для создания загрузочных модулей используются системы программирования с различных языков. Традиционная структура системы программирования представлена на рис. 3.

На этапе кодирования программист с помощью текстового или специализированного для данного языка редактора вводит исходный текст программы. В результате создаются файлы исходных модулей. На следующем шаге компилятор обрабатывает эти файлы, выполняя препроцессорную обработку, лексический, синтаксический и семантический анализ исходного текста, оптимизацию и генерацию кода.

В результате работы компилятора образуются файлы объектных модулей и диагностическая распечатка (листинг). Каждый объектный модуль содержит код, начинающийся с нулевого адреса, и таблицу внешних символов. Таблица внешних символов состоит из таблицы внешних имен и таблицы внешних ссылок (рис. 4). На рис. 5 показан пример двух модулей с внешними именами и внешними ссылками. Переменная A описана в модуле m1, но видна в модуле m2 и других модулях. Это внешнее имя модуля m1. Переменная

B не описана в модуле m1, но видна в нем. Это внешняя ссылка модуля m1. Для модуля m2 A внешняя ссылка, а B внешнее имя. Видимость в модулях внешних имен и внешних ссылок обеспечивают директивы PUBLIC и EXTERN.

Таким образом, таблица внешних имен модуля m1 будет содержать переменную A, а таблица внешних имен модуля m2 будет содержать переменную B. Таблица внешних ссылок модуля m1 будет содержать переменную B, а таблица внешних ссылок модуля m2 будет содержать переменную A.

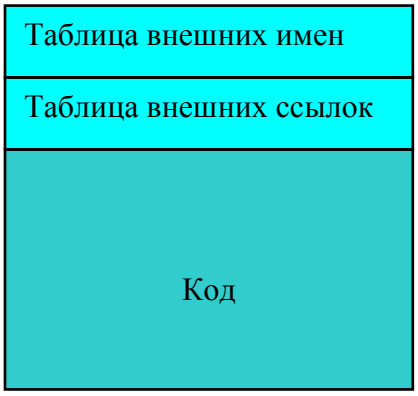


Рис. 4. Структура объектного модуля

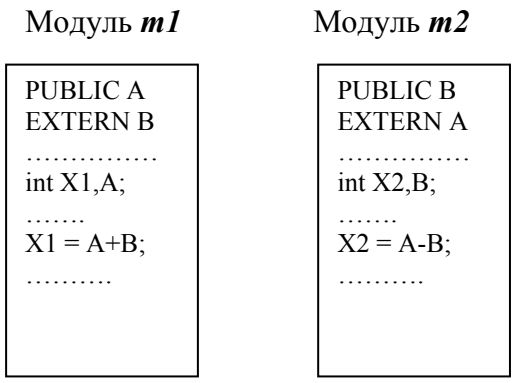


Рис. 5. Внешние имена и ссылки

Файл листинга содержит исходный текст, диагностические сообщения, а также в зависимости от установленных опций может содержать ассемблерный текст (если исходный текст на языке высокого уровня), машинный код модуля.

После того как построены все объектные модули, на следующем шаге выполняется программа компоновщик. Она собирает из построенных объектных модулей, а также из объектных модулей подключенных библиотек загрузочный модуль. Адресация в загрузочном модуле начинается с адреса 0 (нуль). Загрузочный модуль содержит машинный код и таблицу настройки. Эта таблица содержит описания адресов, которые зависят от размещения загрузочного модуля в ОП. Эти адреса называются адресно-зависимыми элементами. Следует заметить, что модуль типа .com в DOS не содержит таблицы настройки, поскольку в нем запрещено использовать подобные адреса. Процесс построения загрузочного модуля простой структуры и его загрузка в ОП рассматриваются далее, в теме «Организация программных модулей».

Также в системы программирования входит такой компонент, как символьный отладчик. Он позволяет вести отладку в терминах входного языка.

Использование рассмотренной традиционной системы программирования имеет серьезный недостаток, связанный с переносимостью ПО. Эта

проблема возникла в 60-70 годах прошлого столетия. Дело в том, что в СССР в то время существовали вычислительные машины серии Минск 2- Минск 22- Минск 32, БЭСМ4- БЭСМ 6, М220-М222, которые существенно отличались своей архитектурой. Для каждой системы делались свои системы программирования с одного и того же языка. В результате одна и та же задача, написанная на языке, скажем FORTRAN, для одной системы существенно отличалась от ее реализации на том же языке для другой системы. Тогда же было найдено решение этой проблемы. Оно состояло в стандартизации некоторого промежуточного языка, для которого создавались интерпретаторы для различных КС. А для исходного языка строился компилятор в этот промежуточный язык. Однако вычислительные возможности машин того времени были не велики, а интерпретационное исполнение программы занимало существенно больше времени, чем скомпилированного загрузочного модуля.

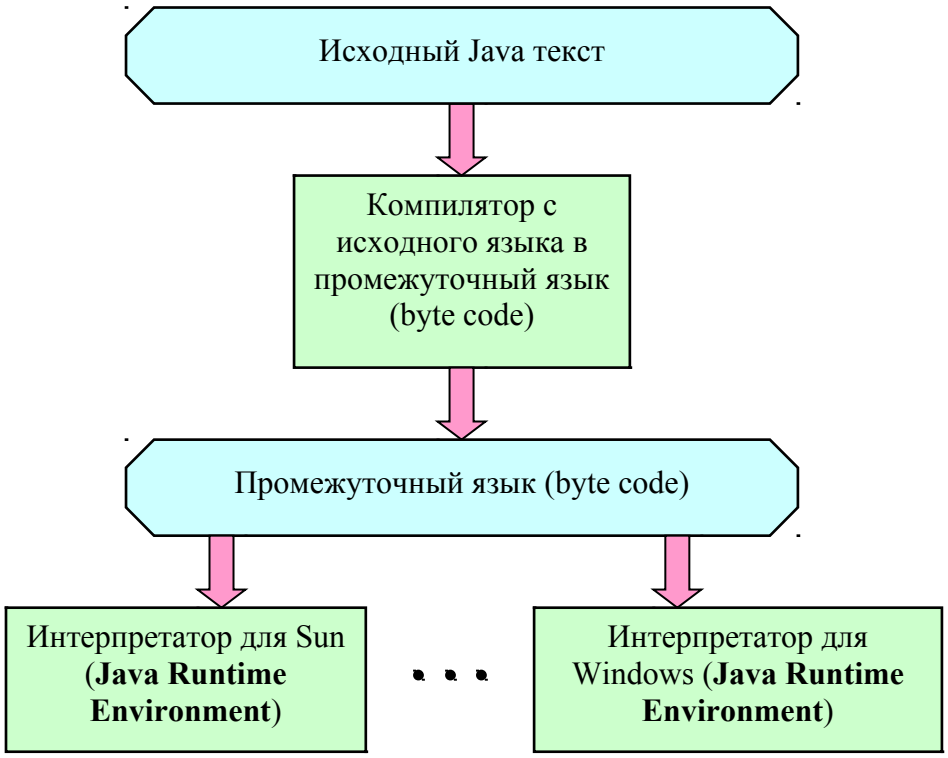


Рис. 6. Структура системы программирования Java

Подобная ситуация сохранилась и до наших дней. Так даже для одной платформы x86 Windows реализация на языке C++ под VisualC++ существенно отличается от реализации под BorlandC++Builder.

Таким образом, проблема состоит в следующем: необходимо построить систему программирования таким образом, чтобы реализация одной и той же задачи на одном и том же языке программирования была одинакова для различных аппаратно-программных платформ.

В настоящее время существуют системы программирования, решающие эту проблему. Пример одной из них – Java. На рис. 6 показана структура этой системы программирования.

Одной из целей разработки системы Java было обеспечения независимости ПО от различных платформ. Разумеется, язык Java является объектно-ориентированным. Компилятор преобразует исходный код в некоторый промежуточный байт-код. Библиотечные модули также хранятся в байт-коде. Для различных платформ разработаны интерпретаторы, которые и исполняют байт-код. Интерпретатор называется Java Runtime Environment (JRE).

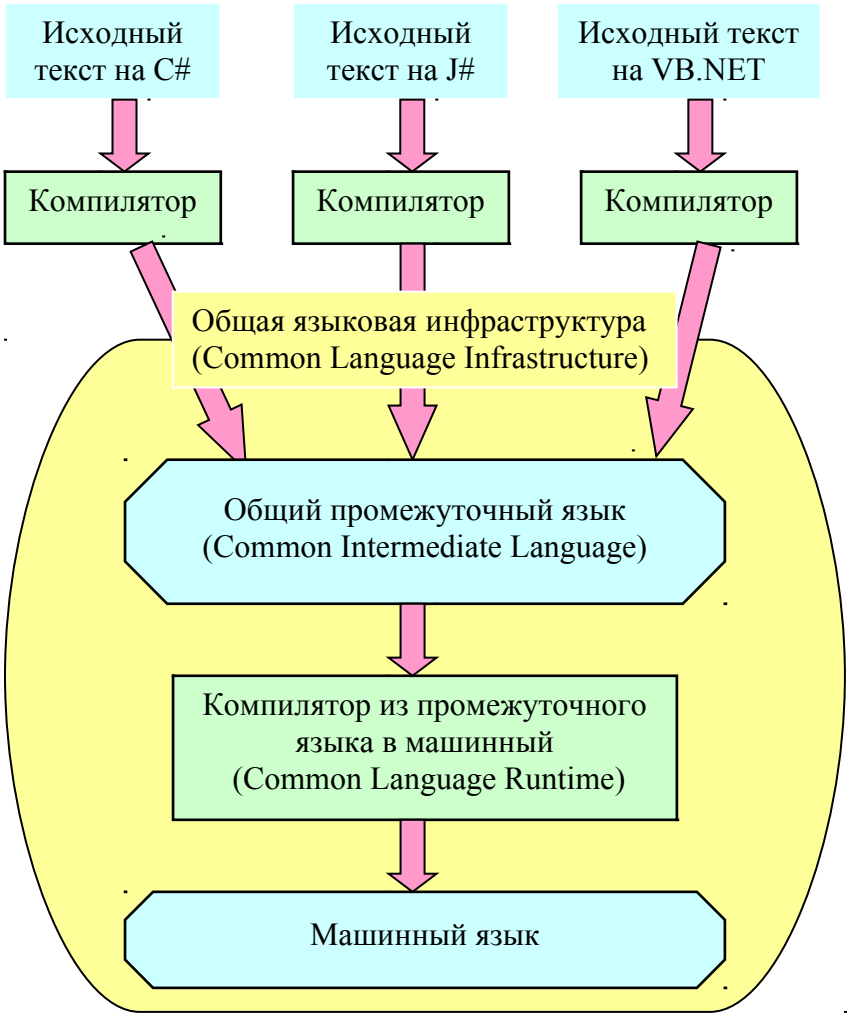


Рис. 7. Структура системы программирования Microsoft .NET Framework

Фирма Microsoft также предприняла шаги в этом направлении, разработав окружение Microsoft .NET Framework – ПО, которое инкапсулируется в ОС Windows и управляет программами, написанными для этого окружения. На рис. 7 представлена структура этой системы программирования.

Существует ряд языков (C#, J#, VB.NET) и компиляторов с этих языков, которые транслируют исходный текст программ на этих языках в общий про-

межуточный язык (CIL - Common Intermediate Language), который вместе с библиотеками классов составляет часть общей языковой инфраструктуры (CLI – Common Language Infrastructure). В CLI также существует компилятор, который транслирует последовательно части кода CIL в машинный язык и исполняет их.

Задание для самостоятельной работы!

Задание 1) Рассмотрите состав версий ОС Windows (Home Edition, Professional, Media Center Edition 2005). Чем они отличаются?

Задание 2) Рассмотрите состав версий Linux.

Задание 3) В той системе, с которой Вы работаете, поставьте опции получения файлов листинга и карты модуля. Для несложного модуля выведите эти файлы и изучите их.

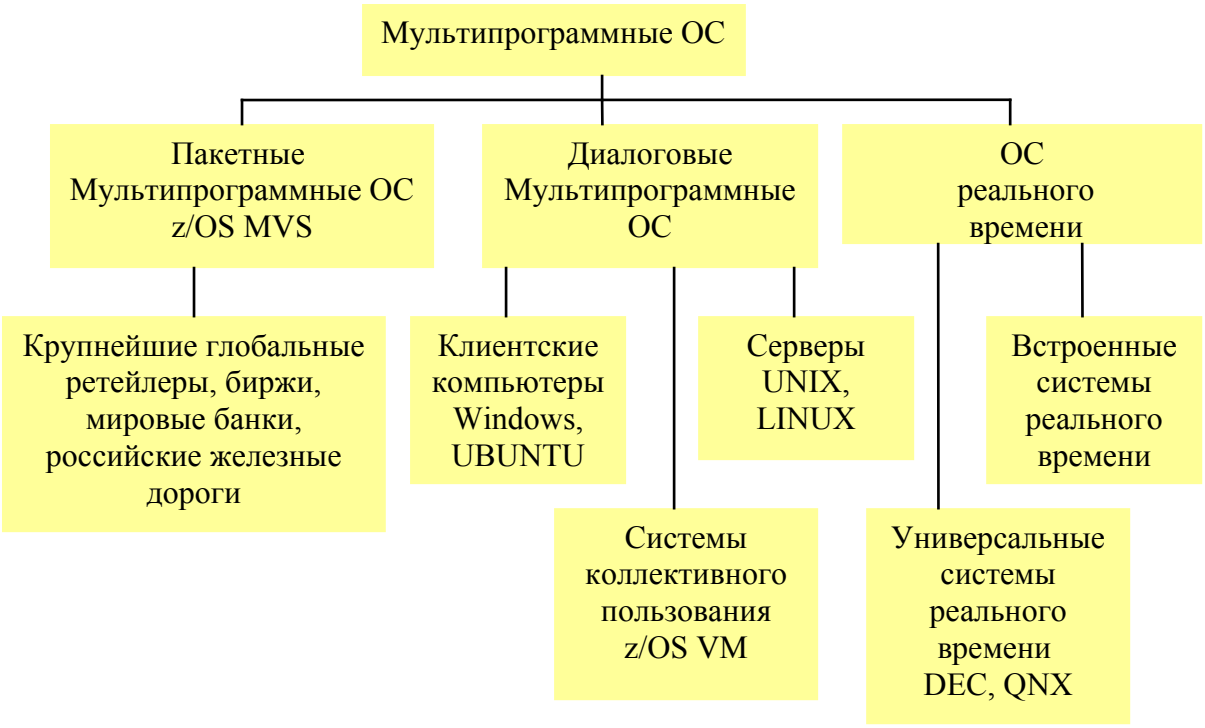
Задание 4) Изучите структуру и организацию системы программирования Java.

Задание 5) Изучите структуру и организацию системы программирования .NET Framework.

Тема 2. ТИПЫ ОС

В этой теме обсуждаются история развития ОС и системы различного типа, ориентированные на различные области применения.

В настоящее время разрабатываются, выпускаются и используются мультипрограммные ОС. Они различаются по типу и областям применения. На рис. представлены основные области применения и типы ОС.



2.1. История развития ОС

В таблице 1 представлены этапы развития ОС.

Таблица 1.

Год	Достижения в области ПО	Тип ОС
1945 – 1950	Библиотеки прикладных программ. Макрогенерация. Ассемблеры.	Отсутствие ОС. Доисторический период
1950 – 1955	Методы доступа. Буферизация. Компоновщики. Перемещающие загрузчики.	
1955 – 1960	Система прерываний и обработчики прерываний. Таймеры. Контрольная точка. Управление файлами.	Появление однопрограммных ОС
1960 – 1965	Классическое мультипрограммирование. Создание системы IBM360 и OS360 MFT	Появление мультипрограммных ОС
1965 – 1975	Динамическое распределение памяти ОС360 MVT. Диалоговые системы. Телеобработка. Виртуальная память IBM370 OS370 SVS, VS1, VS2, MVS. Виртуальные машины VM370.	
1975 – 1980	Сети ЭВМ. Сетевые и распределенные ОС.	Сетевые и распределенные ОС. Мультипроцессорные системы
1980 – 1990	Базы данных. Мультимедиа	
1990 – 2000	Проникновение ВС в повседневную жизнь человека.	Универсальные ОС,

	Создание ОС, имеющих дружественный графический и мультимедийный интерфейс.	поддерживающие мультипроцессорные платформы.
2000 – наши дни	КС охватывают все человечество благодаря гаджетам. Супер компьютеры для научных расчетов. Облачные технологии. Big data.	ОС для гаджетов и суперкомпьютеров

При появлении цифровых вычислительных машин (ЦВМ) и программированию на них выяснилось, что приходится использовать в разных программах одни и те же функции. Поэтому, естественно было создать библиотеку подпрограмм. Первая библиотека в Массачусетском Технологическом институте (Бостон, штат Массачусетс) была оформлена в виде перфокарт, которые лежали в ячейках шкафа. Программисты подкладывали перфокарты нужных функций в свою колоду. Ясно, что такой способ обладал массой недостатков. Поэтому возникла идея макрогенерации и ассемблирования, то есть сборки программы из стандартных фрагментов, которые настраиваются на конкретное применение.

Макрогенерация.

Фрагменты кода оформляются в виде макроопределений. Макроопределение имеет следующую структуру:

```
            MACRO
<метка> <имя макроопределения> <параметры>
            <тело макроопределения>
            ENDM
```

Пример макроопределения:

```
            MACRO
&label          PRINT     &segment,&offset
            mov         AX,&segment
            mov         DS, AX
            mov         AX,&offset
            mov         SI, AX
            mov         AH, 09h
            int         21h
            ENDM
```

Макроопределения располагаются либо в начале ассемблерного текста исходного модуля, либо в библиотеках, которые подключаются при компиляции.

Исходный модуль содержит макрокоманды, которые содержат имя и значения параметров. Для приведенного примера макроопределения, пример макрокоманды выглядит следующим образом:

```
        PRINT           SegmentAdr,OffsetAdr
```

Процесс макрогенерации состоит в следующем: в исходном тексте модуля на место каждой макрокоманды подставляется макрорасширение, которое

образуется из соответствующего макроопределения путем подстановки значений параметров из макрокоманды в текст макроопределения на место соответствующих параметров. Для приведенных примеров макроопределения и макрокоманды макрорасширение выглядит следующим образом:

```
mov     AX, SegmentAdr
mov     DS, AX
mov     AX, OffsetAdr
mov     SI, AX
mov     AH, 09h
int     21h
```

В 60-70 годы было разработано большое количество мощных макрогенераторов. В настоящее время эта идея широко используется в языке C++. Это язык препроцессора и механизм шаблонов.

На следующем этапе появляются методы доступа, которые сочетают структуры данных и способы доступа к данным. Это преддверие систем управления данными и файловых систем. Процессоры становятся более мощными, и для уравнивания скоростей работы внешних устройств и скорости работы процессора появляется буферизация. Совершенствуются методы компоновки программ и загрузки исполнимых модулей в ОП. На смену абсолютным загрузчикам, которые загружали исполнимый модуль с фиксированного постоянного адреса ОП, приходят перемещающие загрузчики, которые позволяют загружать исполнимый модуль с произвольного адреса.

Появление более надежных машин позволило перейти к созданию однопрограммных ОС, но все-таки машины еще не могли работать без сбоев длительное время, а их быстродействие не было столь высоко для возможности решения сложных задач в приемлемое время. Поэтому были разработаны программные средства, названные контрольной точкой. В определенных точках выполнение программы прерывалось, и копия вычислительного процесса сохранялась во внешней памяти. Если в дальнейшем наступал сбой, то возобновление процесса можно было начать с контрольной точки, а не начинать все заново. В современных ОС Windows этот режим поддерживается и называется «Hibernate».

Появление в начале 60-х годов прошлого века системы IBM360 и линейки ОС для нее открыло новую эру развития мультипрограммных ОС. В 60 – 70 годы прошлого века были разработаны основные концепции, принципы организации, методы и алгоритмы ОС.

2.2 Однозадачные ОС

В качестве примеров однозадачных (однопрограммных) ОС можно привести выпущенную фирмой IBM в 1964 году OS360 PCP (prime control

program) и DOS фирмы MicroSoft. OS360 PCP предшествовала серии мультипрограммных ОС и была выпущена для обеспечения возможности продаж системы IBM360. Дело в том, что компьютеры были запущены в производство в 1962 году, а разработка программного обеспечения задержалась на два года. Поэтому для поставки потребителям этой системы и была выпущена более простая версия PCP. Историю КС фирмы IBM можно посмотреть здесь и здесь.

Для работы с этой системой программист должен был составить задание на языке управления заданиями JCL (Job Control Language). Задание состояло из шагов. На каждом шаге выполнялась какая-то программа. Например, типовое задание могло состоять из следующих шагов:

компилировать программу

собрать загрузочный модуль

выполнить загрузочный модуль.

Исходный текст программы и данные для ее выполнения могли быть помещены в задание. Система последовательно вызывала программы, описанные в шагах задания, и выполняла их. Структура такой системы представлена на рис. 8.

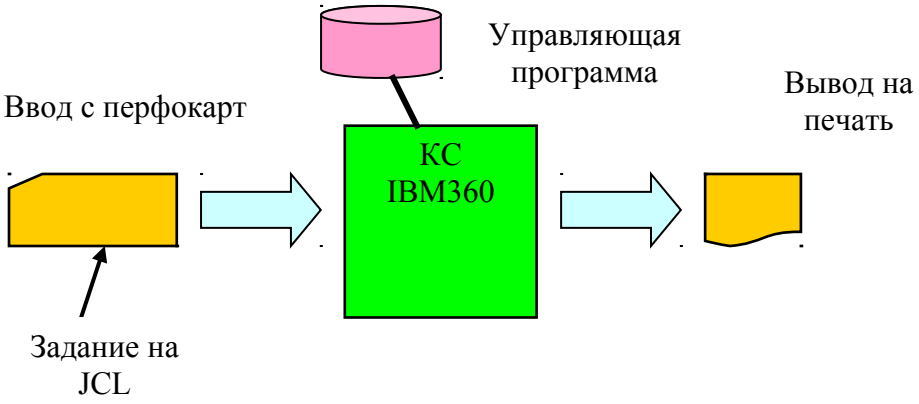


Рис. 8. Структура однопрограммной системы

Другой пример однопрограммной ОС – MS DOS v.6.22, которая сейчас эмулируется версиями Windows. Разумеется, на смену перфокартам и печати пришли ввод с клавиатуры и вывод на экран монитора, то есть общение стало диалоговым. Вместо ввода всего задания, написанного на JCL, пользователь вводит команды динамически, командный процессор разбирает очередную команду и организует ее выполнение, хотя возможность задать выполнение последовательности команд осталась в виде командных файлов.

Состав однопрограммной системы:

процессор командного языка, который выполняет команды и командные файлы,

система управления внешними устройствами,

файловая система.

Характерные свойства однопрограммной ОС:

- в системе выполняется одна задача, поэтому все ресурсы отдаются этой задаче и когда она заканчивается, ресурсы освобождаются,

- переключение между задачами осуществляет пользователь.

2.3 Пакетные ОС

Производительность КС росла, и на них решались все более сложные задачи. Характер этих задач таков, что требуется осуществление однотипных расчетов для потока данных или результаты расчетов могут быть получены с определенной задержкой. Такие задачи называются пакетными в отличие от диалоговых или задач реального времени.

При обработке задач такого типа существенным является повышение производительности, измеряемое в числе решенных задач в единицу времени. И первым средством повышения производительности было исключение человека из процесса обработки, то есть переключение между задачами происходит автоматически.

Пример такой системы был реализован еще до появления системы IBM360 на более ранних машинах этой фирмы. Весь процесс обработки осуществлялся системой подготовки данных и расчетной системой.

Система подготовки данных (рис. 9) была реализована на маломощной машине IBM1401. Задания подготавливались на перфокартах и вводились в машину, а система подготавливала поток заданий на магнитной ленте (МЛ). Затем эта МЛ переносилась на более мощную расчетную систему. Результаты обработки также писались расчетной системой на магнитную ленту (МЛ) и переносились для распечатки на маломощную систему.

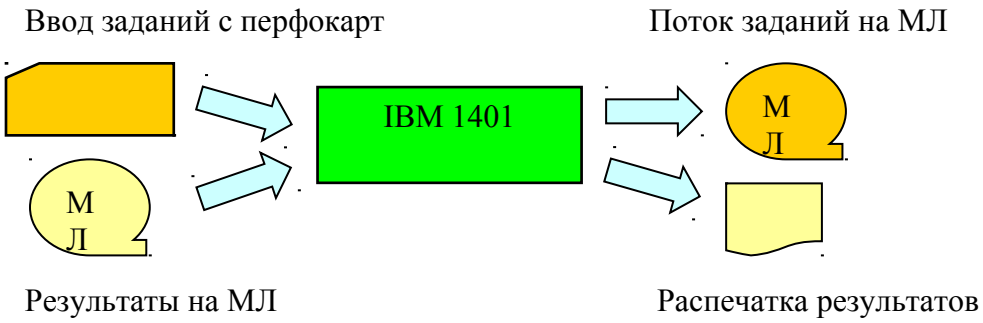


Рис. 9. Система подготовки данных

Расчетная система (рис. 10) принимала поток заданий на МЛ, считывала по очереди задания и последовательно шаг за шагом выполняла их. Результаты выполнения заданий писались на МЛ.

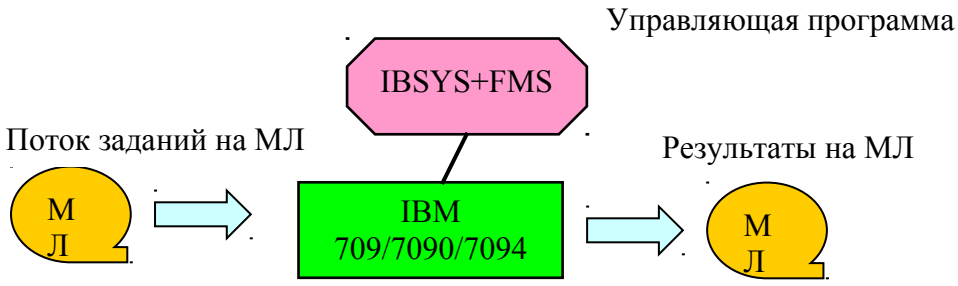


Рис. 10. Расчетная система

Состав пакетной системы:

управление потоком заданий,

система управления внешними устройствами,

файловая система.

Характерные свойства пакетных ОС:

- в системе выполняется одна задача, поэтому все ресурсы отдаются этой задаче и когда она заканчивается, ресурсы освобождаются,

- переключение между задачами осуществляет управляющая программа.

2.4. Мультипрограммные пакетные ОС

Примерами мультипрограммных пакетных ОС могут служить серия OS360 для IBM360. Это версии MFT, MVT. Системы виртуальной памяти для IBM370: SVS, VS1, VS2, MVS, а также мэйнфреймы z-series с z/OS MVS.

Создание мультипрограммных пакетных ОС преследовало цель еще более повысить производительность систем, выполняющих пакетную обработку. Следующим шагом после исключения человека из процесса обработки стало создание КС, способной обеспечить одновременное выполнение нескольких задач, разделяющих устройства системы. Для осуществления этой идеи была спроектирована IBM360.

В ходе своего выполнения каждая задача занимает процессор, а также осуществляет операции ввода/вывода. По сравнению со скоростью выполнения команд процессором операции ввода/вывода достаточно медленные. Если поручить управление этими операциями периферийному процессору, освободив, таким образом, центральный процессор (ЦП), то на ЦП может выполняться другая задача. На рис. 11 показана структура такой системы.

Цель, которая преследуется при создании ОС для такой архитектуры, состоит в построении механизмов управления, которые обеспечивают максимальную загрузку устройств КС и, прежде всего, ЦП.

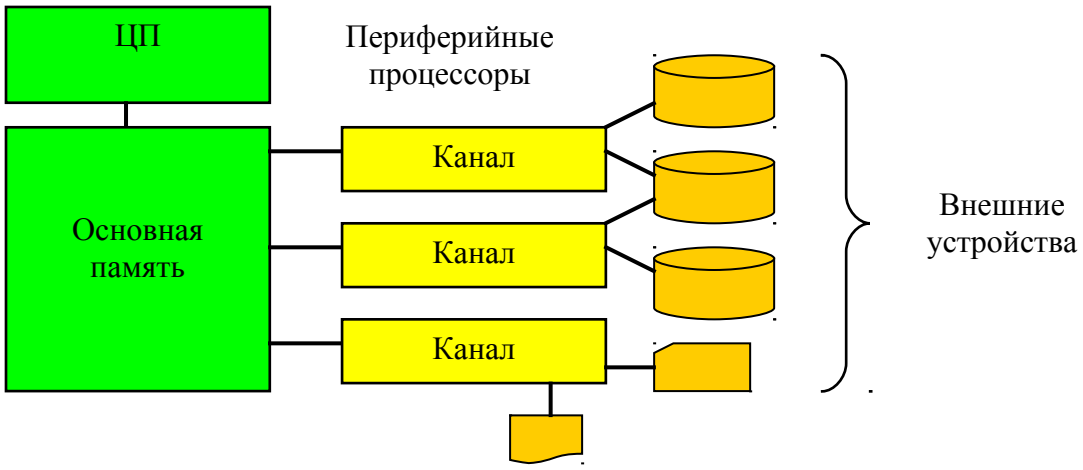
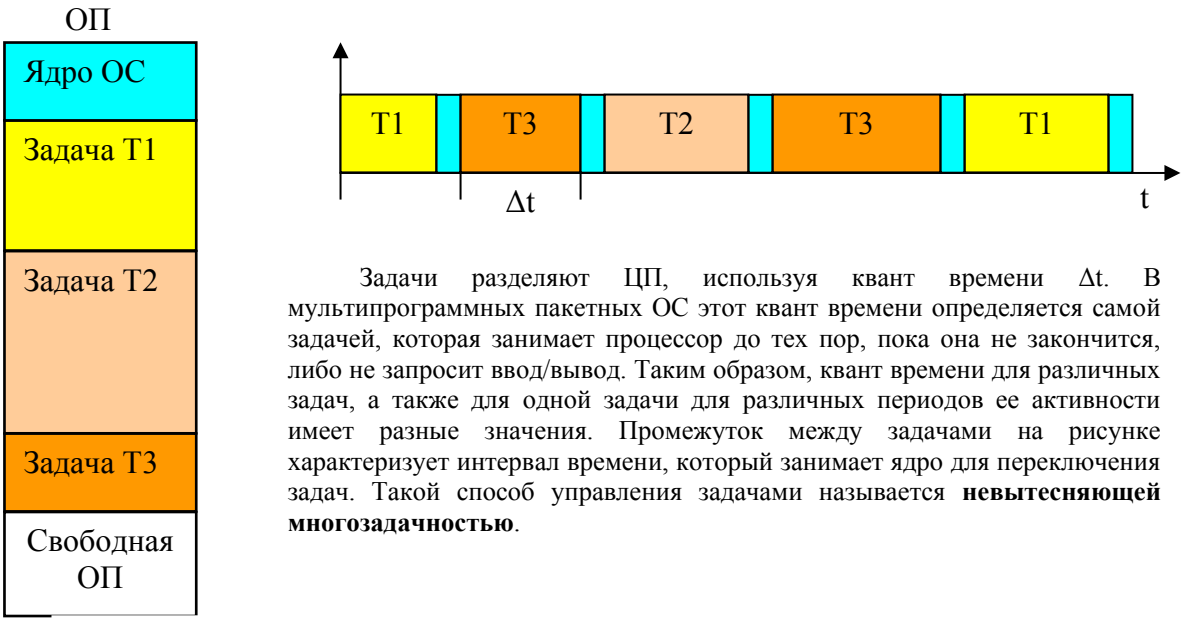


Рис. 11. Структура аппаратной части мультипрограммной пакетной КС

Аппаратные средства такой системы, кроме периферийных процессоров и соответствующих интерфейсов, имели развитую систему прерываний и защиту ОП и устройств от проникновения одних задач в области полномочий других задач, а также в системную область памяти. Процессор имел два режима работы: режим супервизора и режим прикладной задачи. В режиме супервизора выполнялись все команды, а в режиме прикладной задачи некоторые команды недопустимы. Например, команды, связанные с вводом/выводом. Таким образом, прикладные задачи для осуществления ввода/вывода должны обращаться к управляющей программе. На базе таких аппаратных средств были разработаны мультипрограммные пакетные системы.

Алгоритм управления задачами в системах такого типа основан на принципе не вытесняющей многозадачности. Этот принцип состоит в следующем: если какая-либо задача (процесс) выполняются на ЦП, то эта задача освободит процессор либо когда она закончится, либо когда она обратиться к управляющей программе с каким-либо запросом, например, запросом на ввод/вывод. Иногда в литературе для обозначения такого принципа управления используют запись $\Delta t \to \infty$. Это следует понимать так: управляющая программа отводит задаче квант процессорного времени Δt бесконечно большой, она не собирается прерывать эту задачу.

Мультипрограммирование – такой способ управления задачами в ОС, который позволяет двум или более задачам в процессе выполнения занимать определенную часть ОП, разделять время ЦП и другие устройства ВС. На рис. 12 показана структура ОП, в которую загружены три задачи и разделение времени ЦП.



Задачи разделяют ЦП, используя квант времени Δt. В мультипрограммных пакетных ОС этот квант времени определяется самой задачей, которая занимает процессор до тех пор, пока она не закончится, либо не запросит ввод/вывод. Таким образом, квант времени для различных задач, а также для одной задачи для различных периодов ее активности имеет разные значения. Промежуток между задачами на рисунке характеризует интервал времени, который занимает ядро для переключения задач. Такой способ управления задачами называется **невытесняющей многозадачностью**.

Рис. 12. Структура ОП и разделение времени ЦП

С мультипрограммным режимом работы связаны такие понятия как уровень мультипрограммирования и смесь задач.

Уровень мультипрограммирования – число задач, выполняющихся одновременно в КС.

Смесь задач – качественная характеристика состава задач, выполняющихся одновременно, которая влияет на загрузку ЦП. Эта характеристика связана со свойствами выполняемых задач. Это свойство определяется процентом времени выполнения задачи, отводимого на операции с внешними устройствами, то есть все время выполнения задачи можно разделить на ту часть, когда задача занимает ЦП, и часть, когда она занимается вводом/выводом и освобождает ЦП. Понятие смеси задач поясним на примере двух задач, которые выполняются в мультипрограммном режиме. Пусть одна из этих задач занимается расчетами, то есть мало времени тратит на ввод/вывод, а много времени занимает ЦП. Назовем эту задачу счетной. Другая задача наоборот, занимает мало времени ЦП, но активно работает с внешними устройствами. Назовем эту задачу информационной. Выполнение этих двух задач позволит загрузить и ЦП, и внешние устройства, поскольку счетная задача будет иметь возможность занимать ЦП в те интервалы времени, когда информационная задача осуществляет работу с внешними устройствами. Главное обеспечить информационной задаче возможность занимать центральный процессор своевременно для запуска очередных операций с внешними устройствами.

Если бы в смеси присутствовали только счетные задачи, то внешние устройства бы простаивали, а процессор был бы загружен. Если бы в смеси

присутствовали только информационные задачи, то ЦП бы простаивал. Таким образом, формирование оптимальной смеси задач улучшает производительность ВС.

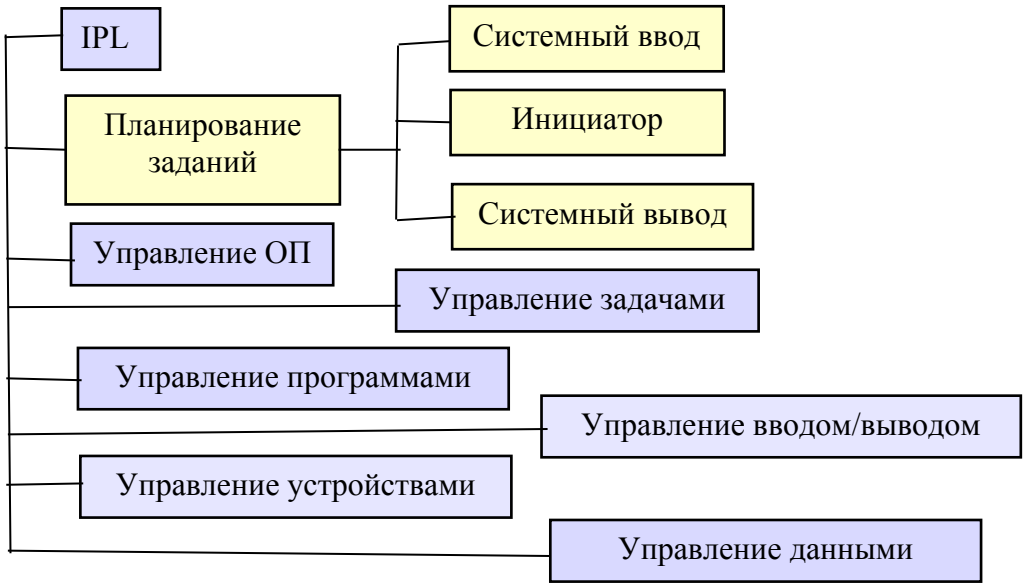


Рис. 13. Структура управляющей программы пакетной мультипрограммной ОС

На рис.13 показана структура управляющей программы. В мультипрограммной пакетной системе существуют два уровня управления. Верхний уровень состоит из системных задач, которые стартуют в системе как привилегированные процессы. В современной z/OS они называются JES2 или JES3 (Job Entry System). Основная задача верхнего уровня состоит в организации прохождения заданий через систему. Для этого необходимо провести разбор поступившего задания и перевести операторы языка JCL в совокупность связанных таблиц. Затем по этим таблицам производится подготовка запуска прикладной программы, ее запуск и обработка завершения.

Все задачи как системные, так и прикладные используют функции ядра. Программы ядра резидентно размещаются в памяти системы, и доступ к ним происходит с помощью программного интерфейса.

Программа начальной загрузки IPL (Initial program loader) присутствует во всех системах и служит для загрузки ОС при старте компьютера. Это весьма короткая программа, которая загружает с определенного места диска другую программу, которая собственно и занимается загрузкой ядра системы.

2.4.1 Планировщик заданий

В первых системах корпорации IBM планировщик заданий состоял из трех компонентов. Это задачи системного ввода, которых могло быть до трех, в зависимости от количества устройств ввода заданий. Задачи инициа-

ции/завершения управляют созданием и завершением прикладных задач. Их могло быть до 15. Задачи системного вывода управляют выводом информации на различные устройства, и их могло быть до 36. Все задачи планировщика заданий являются системными и размещаются в той же области ОП, что и прикладные задачи. Планировщик заданий используется только в пакетных системах. В других типах ОС его использование не имеет смысла, поскольку нет потока заданий, который он обрабатывает.

В настоящее время примером такой системы является z/OS MVS. Существует два планировщика заданий JES2 и JES3, которые теперь называются Job Entry System.

Следующие компоненты управляющей программы являются функциями ядра и располагаются резидентно в ОП. Это функции управления ОП, задачами, внешними устройствами, вводом/выводом и данными. Подобные функции существуют и в других типах ОС, но реализуют другие алгоритмы.

Планировщик заданий обрабатывает входной пакет заданий и управляет прохождением заданий через КС. Пакет заданий состоит из отдельных последовательно поступающих заданий и заканчивается признаком конца – два символа “//”. Пример текста на языке управления заданиями (JCL – job control language) приведен ниже.

```
//task1                    JOB   <параметры оператора JOB>
//                         EXEC  PGM=FORTRAN <другие параметры оператора STEP>
//<имя DD оператора>       DD    <параметры DD оператора>
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
//<имя DD оператора>       DD    <параметры DD оператора>
//                         EXEC  PGM=LINK <другие параметры оператора STEP>
//<имя DD оператора>       DD    <параметры DD оператора>
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
//<имя DD оператора>       DD    <параметры DD оператора>
//                         EXEC  GO=*
//SYSIN                    DD    *
              <Входные данные программы>
//task2                    JOB   <параметры оператора JOB>
 . . . . . . . . . . . . . . . .<описание  второго задания>. . . . . . . . . . . .
//                                                                  . . . . . . .
```

Оператор JOB определяет начало описания очередного задания. Этот оператор содержит параметры для всего задания. Например, оператор CLASS=A описывает класс, в котором будет выполняться это задание.

Оператор EXEC задает программу, которая будет выполняться на этом шаге, а также параметры для этого шага. Операторы DD (data definition) определяют устройства и наборы данных, которые использует программа. Так на первом шаге будет выполняться программа FORTRAN – компилятор с языка FORTRAN. На втором шаге выполняется LINK, а на третьем выполня-

ется изготовленный загрузочный модуль. Далее идут другие задания, входящие в пакет.

Пакет заданий мог размещаться на перфокартах, на магнитной ленте или на магнитном диске. Когда появились средства диалогового ввода, то пользователь стал вводить задание с клавиатуры, а далее задание поступало на обработку стандартным образом.

Стандартная обработка задания состоит в следующем. Задача Системного Ввода вводит задание в КС, производит разбор текста задания и записывает список таблиц задания в очередь входных работ определенного класса (рис. 14).

JCT (job control table) – таблица управления заданием. Содержит параметры оператора JOB. Одна таблица на задание. SCT (step control table) – таблица управления шагом задания. Содержит параметры оператора EXEC. Строится для каждого шага задания. SIOT (step input output table) – таблица ввода вывода шага задания. Содержит описания внешних устройств используемых в DD операторах шага задания. JFCB (job file control block) – таблица управления файлом задания. Содержит описания наборов данных в DD операторах шага задания. SIOT и JFCB создаются для каждого DD оператора. Эта информация позволяет инициатору аллокировать наборы данных для задачи шага задания предварительно до запуска задачи.

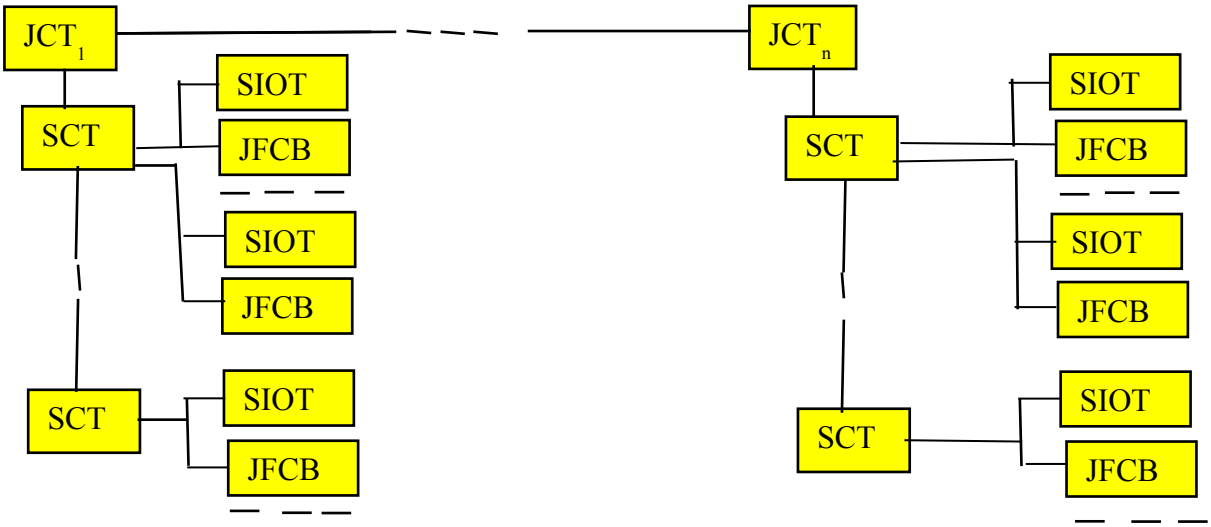


Рис. 14. Очередь входных работ

Все очереди входных работ имеют следующую структуру (Рис. 15). Каждое задание принадлежит определенному классу, который задается параметром CLASS в операторе JOB. Классы обозначаются латинскими буквами. Каждый класс входных работ характеризуется определенным объемом ОП, предоставляемым заданиям и временем выполнения. Таким образом, задачи шагов задания различных классов формируют смесь задач, выполняемых в мультипрограммном режиме. Задачи одного задания, а также задания одного

класса выполняются последовательно. Список таблиц задания (СТЗ) ставится в очередь последовательно.

Очередь входных работ каждого класса обрабатывает задача инициации/завершения этого класса.

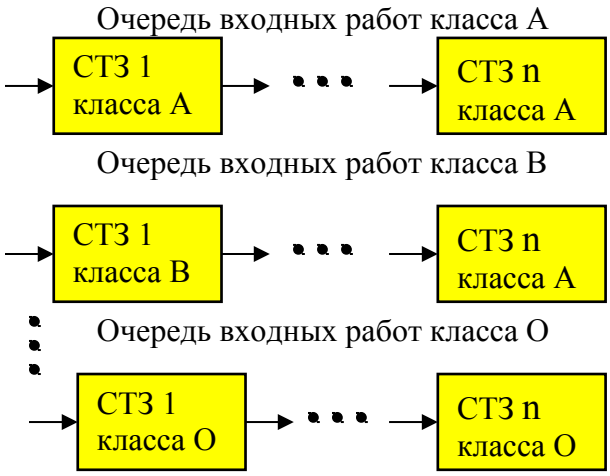


Рис. 15. Структура очередей входных работ различных классов

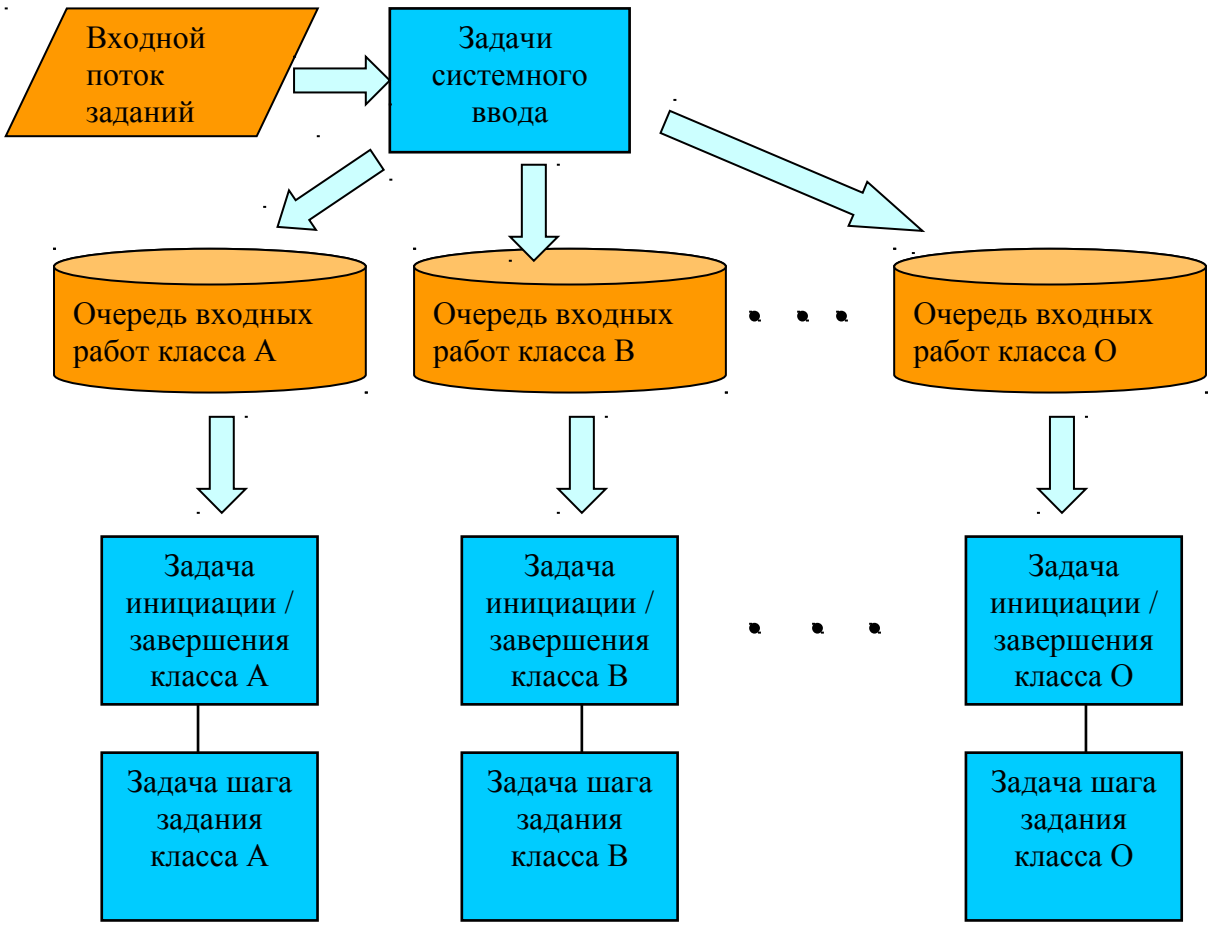


Рис. 16. Структура обработки входного потока заданий

Задача инициации/завершения выбирает очередное задание из очереди входных работ данного класса и последовательно обрабатывает шаги этого задания, подготавливая для очередного шага выполнение программы шага задания. Подготовка состоит в запросе раздела ОП, размещения наборов данных и требуемых ресурсов для задачи шага задания. После того, как функции ядра отработают эти запросы и выделят требуемые ресурсы, задача инициации/завершения создаст задачу для шага задания, в рамках которой выполняется программа шага, и передаст ей управление. После завершения задачи шага задания, управление опять получает задача инициации/завершения, которая закрывает незакрытые программой шага задания файлы, уничтожает предписанные файлы, освобождает использованные ресурсы, освобождает участок ОП и выводит информацию о выполнении задачи шага задания. Схема функционирования задач планировщика при обработке очереди входных работ показана на рис. 16. Разумеется, в реальных системах все 15 классов использовались в редких случаях

Задачи системного ввода, инициации/завершения и шага задания выводят различную информацию в процессе работы. Задачи выполняются в различные промежутки времени, разделяя время ЦП, а выходная информация, вырабатываемая в процессе выполнения определенного задания, должна выводиться слитно. Для решения этой проблемы используется виртуальное устройство вывода, которое называется очередью выходных работ.

В системе могло быть до 36 очередей выходных работ, которые обозначались двадцатью шестью латинскими буквами (A,…,Z) и десятью арабскими цифрами (0,…,9). Так, например, очередь A связывалась с принтером, а очередь B с выводом на перфокарты. С появлением систем телеобработки, удаленных систем, вывод на эти системы связывался с очередями выходных работ.

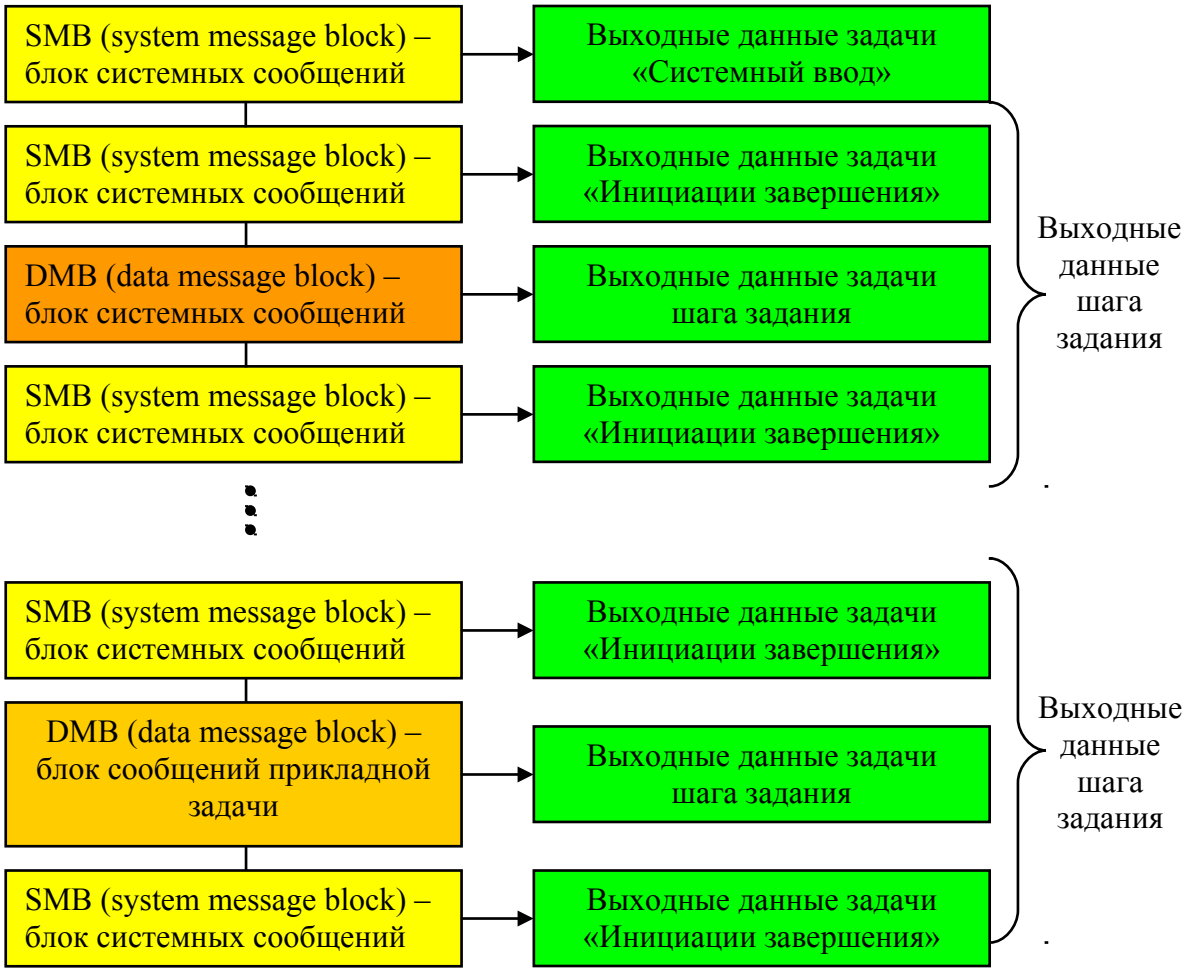


Рис. 17. Список управляющих блоков вывода для одного задания.

Каждая очередь выходных работ представляет собой последовательность списков вывода. Каждый список вывода состоит из управляющих блоков и относится к одному заданию (рис. 17). Выходные данные для каждого задания формируются задачей системный ввод, а также задачами инициации/завершения и шага задания для каждого шага задания. Задания различных классов входных работ выполняются параллельно и могут выводить информацию в один и тот же класс выходных работ. Поэтому вначале формируется список вывода для задания, а затем он вставляется в очередь класса выходных работ (рис. 18).

Собственно это было одним из первых применений виртуальных устройств. Задачи осуществляли стандартный ввод/вывод на устройства, но реально информация поступает в специальные наборы данных и только системной программой вывода будут направлены на реальное устройства.

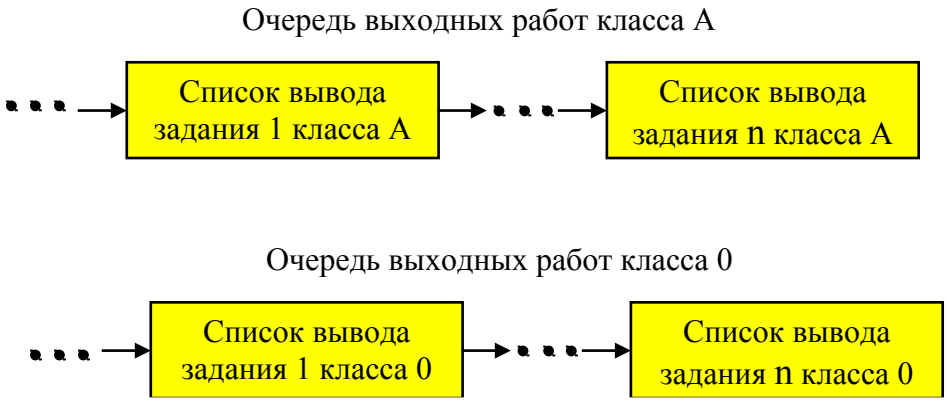


Рис. 18. Очереди выходных работ.

Задачи планировщика обслуживают прохождение программ в системе. Они подготавливают ресурсы, необходимые задаче шага задания, запускают эту задачу и обрабатывают ее завершение. Это возможно благодаря предварительному описанию каждой задачи в задании на JCL.

2.5 Диалоговые мультипрограммные ОС

Идея создания диалоговых ОС возникла примерно в то же время что и пакетной обработки, но отсутствие долгое время внешних устройств для визуального отображения информации сдерживало появление систем этого типа. К диалоговым мультипрограммным ОС следует отнести UNIX, Linux, Windows и систему виртуальных машин VM370 фирмы IBM.

Цель создания диалоговых ОС – облегчить доступ пользователей к КС в начальный период существования таких систем, сделать его комфортным и эффективным в дальнейшем. Поэтому при разработке диалоговых мультипрограммных ОС необходимо обеспечить приемлемую реакцию системы на запросы пользователей. У каждого пользователя должно создаваться впечатление, что он один владеет ресурсами КС, он не должен чувствовать присутствие других пользователей.

К числу таких систем относятся системы коллективного пользования (рис. 19). Такие системы представляли собой достаточно производительную ВС, к которой подключены удаленные терминалы пользователей. Каждый пользователь, имеющий доступ к системе имел имя и пароль, и мог с любого терминала использовать КС.

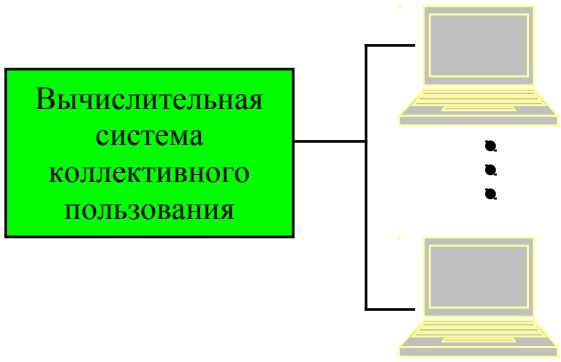


Рис. 19. Диалоговая система коллективного пользования.

С развитием средств вычислительной техники фирма IBM использовала подобную схему в разработке систем продаж для торговых фирм. Несколько десятков тысяч торговых агентов разъезжают по всему миру, рекламируя и продавая продукцию фирмы. Каждый агент должен был иметь каталоги товаров и прочую атрибутику для своей деятельности. Фирма IBM предложила использовать так называемых «тонких клиентов», которые представляли собой, говоря современным языком, планшеты, не имеющие жесткого диска. Тонкий клиент подключался по сети к серверам на базе мощных мэйнфреймов и отображал необходимую информацию. Кроме того, таким образом можно было, и оформить сделку. Такая технология продаж позволила не только существенно повысить качество, но и ускорить время осуществления сделок, что позволило резко увеличить объем продаж.

В современном состоянии к диалоговым мультипрограммным ОС относятся ОС серверов, клиентских станций. Действительно, под их управлением выполняются web-сервера, интерпретаторы PHP и PERL, которые выполняют скрипты, написанные разработчиками сайтов, а также CGI-приложения.

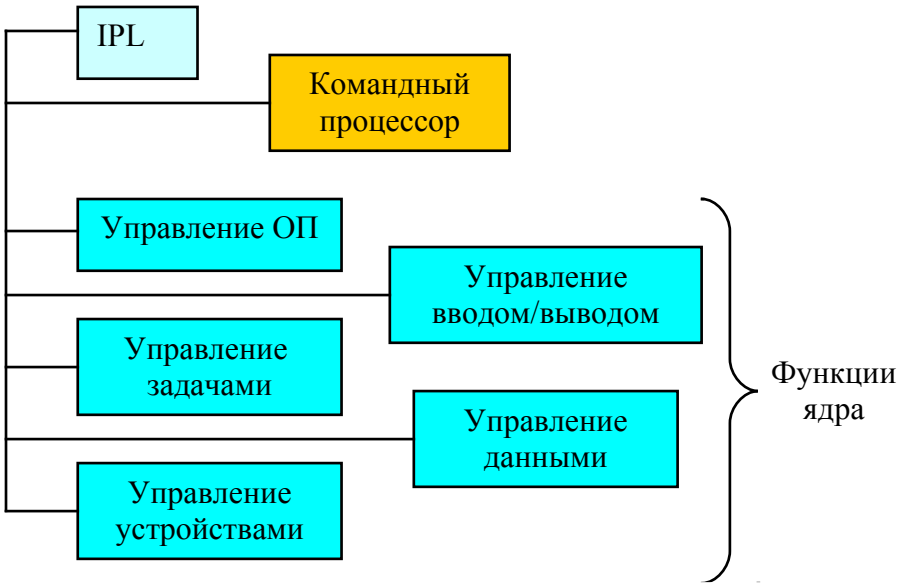


Рис. 20. Структура управляющей программы мультипрограммной диалоговой ОС

Учитывая, что диалоговые ОС должны обеспечивать хорошую реакцию системы, управляющая программа в отличие от мультипрограммных пакетных ОС, определяет квант времени Δt. Для этого используется таймер, в который перед запуском приложения загружается значение интервала времени, на которое задаче предоставляется процессор. По истечению этого интервала возникает прерывание от таймера, и управляющая программа переключает процессор на другую задачу, также определяя для этой задачи значение интервала времени. Такой способ управления, когда управляющая программа

определяет промежуток времени для задач, в течение которого они занимают процессор, называется вытесняющей многозадачностью.

Структура управляющей программы мультипрограммной диалоговой ОС показана на рис. 20. В диалоговых ОС, в отличие от пакетных систем, неизвестна информация о последовательности выполняемых пользователем программ и их описание отсутствует. В каждый момент времени запускается некоторая задача, и ее описание известно ОС в этот момент времени. Поэтому надобность в планировщике заданий отсутствует, а необходимо присутствие командного процессора, который обрабатывает и исполняет команды пользователя.

2.6 Операционные системы реального времени

ОС реального времени отличаются от других мультипрограммных систем тем, что число задач, выполняемых под их управлением, фиксировано. Системы реального времени (СРВ) предназначены для управления различными физическими объектами. В качестве таких объектов могут выступать ядерные и термоядерные реакторы, радиолокационные станции обнаружения и слежения, системы взлета и посадки самолетов или других летательных объектов и множество других объектов. Даже когда вы покупаете билеты на поезд или самолет, вы сталкиваетесь с СРВ, которая занимается резервированием билетов.

Общую схему СРВ можно представить (рис. 21) в виде взаимодействия физического объекта и ВС.

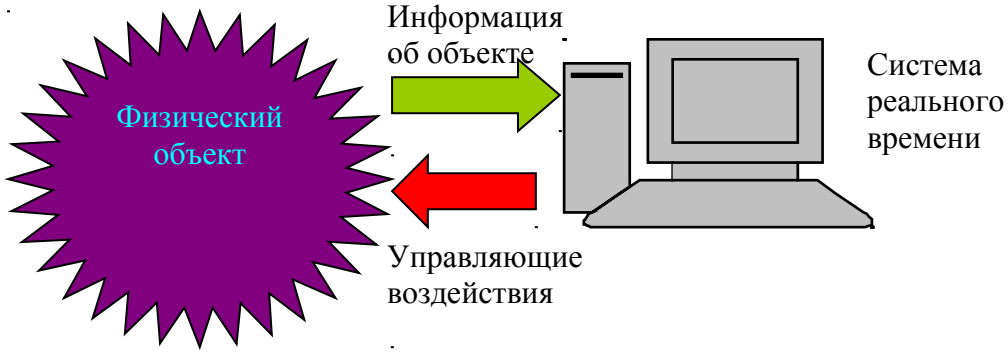


Рис. 21. Общая схема системы реального времени

Информация о физическом объекте поступает в ВС. Там она обрабатывается, и формируются управляющие воздействия. Для СРВ важно, чтобы время ответа на определенное воздействие не превышало определенной величины, то есть время реакции системы ограничено и не может превышать некоторого значения. Поэтому в СРВ используется такое понятие как масштаб времени, которое характеризует допустимое время реакции систе-

мы. Ясно, что требования ко времени реакции СРВ, управляющей летательными объектами существенно жестче, чем в системе резервирования билетов.

С точки зрения масштаба времени ОС реального времени (ОСРВ) можно разделить на два типа:

Нарушение масштаба времени недопустимо и приводит к фатальным последствиям.

Нарушение масштаба времени не приводит к фатальным последствиям, а снижает их эффективность.

В первом случае речь идет о системах управления сложными техническими объектами. В качестве примера систем второго типа можно привести ту же систему резервирования билетов.

Характерные свойства ОС РВ.

Множество внешних событий известно и можно иметь полный набор функциональных программ с изученными характеристиками.

Поток событий, приходящих на вход системы, частично детерминирован во времени. Существуют объекты, требующие периодического обслуживания с постоянным или мало колеблющимся периодом обслуживания.

Входные и выходные данные имеют относительно простую структуру, а периферийные устройства простые, но специфические методы доступа.

Все функциональные программы считаются отлаженными. Известны их характеристики по быстродействию и занимаемой памяти.

С точки зрения конструкции СРВ можно выделить также два типа систем:

Встроенные системы (embedded system), которые специально разрабатываются для определенной задачи. Как правило, это связано с управлением в жестких временных ограничениях, либо в жестких условиях эксплуатации на борту летательных, плавающих объектов.

Универсальные СРВ, которые используют серийно выпускаемые КС и ОС. Прежде всего, следует отметить специализирующуюся в этом направлении фирму Digital Equipment Co. (DEC), которая выпускала целую серию ОСРВ для шестнадцати разрядных ВМ PDP11, затем для 32 разрядной VAX ОС VMS, которая существует и в настоящее время на процессорах фирмы Motorolla. Кроме того, следует отметить распределенную ОСРВ QNX.

Тема 3. ПРИНЦИПЫ ПОСТРОЕНИЯ МУЛЬТИПРОГРАММНЫХ ОС

В первых ОС управляющая программа представляла собой по своей логической структуре нечто напоминающее блюдо спагетти. Опыт разработки OS360 поставил вопросы создания технологии и соблюдения определенных принципов построения ОС. Вот эти принципы, которым отвечают современные ОС.

3.1 Принцип модульности

Программный комплекс обладает свойством модульности, если он состоит из программных единиц, каждая из которых является:

1) функционально самостоятельной,

2) имеет стандартную структуру,

3) стандартным образом связывается с другими программными единицами.

Соблюдение принципа модульности придает ОС следующие свойства:

1) Удобство эксплуатации, которое позволяет быстро осваивать ОС, улучшает ее познаваемость, обеспечивает удобное сопровождение.

2) Возможность развития, что связано с возможностью пополнения ОС новыми компонентами, например, драйверами новых устройств. Это свойство делает ОС открытой системой.

3) Гибкость, которая позволяет эксплуатировать ОС на различных аппаратных конфигурациях ВС.

Функциональная самостоятельность модулей предполагает выполнение модулем одной функции. С этим тесно связано понятие функциональной прочности. Не следует строить систему, размазывая одну функцию по нескольким модулям. С другой стороны, нельзя нагружать модуль несколькими функциями.

Стандартная структура загрузочных модулей жизненно необходима для работы ОС, поскольку загрузчик требует определенного формата загрузочного модуля. Драйверы устройств также должны иметь стандартное описание для подключения в ОС.

Загрузочные модули хранятся на внешних носителях в определенном формате. Файл загрузочного модуля в общем случае состоит из управляющей информации для загрузчика и, собственно, исполнимого модуля.

Управляющая информация для загрузчика представлена в таблице настроек (Relocation Table). В качестве примера рассмотрим структуру этой таблицы для модуля .EXE. Таблица располагается в начале файла. Затем идет

код исполнимого модуля, который является копией образа памяти этого модуля. Это означает, что адреса сегментов выровнены по границе параграфа, то есть, кратны 16, и стек также присутствует в файле. Существует другой подход в системе UNIX, когда стек и куча в файле отсутствуют, а строятся в момент загрузки в основную память как динамический сегмент.

Начало таблицы настройки имеет вид, показанный в табл. 2.

Таблица 2

Смещение	Размер	Содержимое поля
00h	2	4D5A - "подписъ" компоновщика, указывающая на то, что файл является файлом EXE.
02h	2	Длина образа задачи по модулю 512 (т.е. число полезных байт в последнем блоке). (Компоновшики версий до 1.10 помещали в это поле 04; если оно имеет такое значение, его рекомендуется игнорировать).
04h	2	Длина файла в блоках.
06h	2	Число элементов таблицы настройки адресов.
08h	2	Длина заголовка в 16-ти байтных параграфах. Используется для выяснения начала тела загрузочного модуля.
0Ah	2	Минимальный объем памяти, которую нужно выделить после конца образа задачи. (в 16-ти байтных параграфах)
0Ch	2	Максимальный объем памяти, которую нужно выделить после конца образа задачи. (в 16-ти байтных параграфах).
0Eh	2	Сегментный адрес начала стекового сегмента относительно начала образа задачи.
10h	2	Значение SP при входе в задачу.
12h	2	Контрольная сумма - ноль минус результат сложения без переноса всех слов файла.
14h	2	Значение IP (счетчика команд) при входе в задачу.
16h	2	Сегментный адрес начала кодового сегмента относительно начала образа задачи.
18h	2	Адрес первого элемента настройки адресов относительно начала файла.
1Ah	2	Номер сегмента перекрытий. (0 для корневого сегмента программы).

Эту часть таблицы часто называют стандартной частью заголовка.

Далее следует информация для настройки адресов. Таблица состоит из элементов, число которых записано в байтах 06-07. Элемент настройки состоит из двух полей: 2-х байтного смещения и 2-х байтного сегмента, и указывает слова в загрузочном модуле, содержащие адрес, который должен быть настроен на место памяти, в которое загружается кодовый сегмент программы.

При загрузке в основную память функция управляющей программы (Загрузчик) выполняет следующие действия по настройке программы на адресное пространство, отведенное этой программе:

В области памяти после резидентной части выполняющей загрузку программы строится Префикс Программного сегмента (PSP).

Стандартная часть таблицы считывается в рабочую область Загрузчика.

Определяется длина тела загрузочного модуля (разность длины файла 04-05 и длины заголовка 08-09 плюс число байт в последнем блоке 02-03). В зависимости от признака, указывающего загружать задачу в конец памяти или в начало, определяется сегментный адрес для загрузки. Этот сегмент называется начальным сегментом.

Загрузочный модуль считывается в начальный сегмент.

Таблица настройки после стандартной части порциями считывается в рабочую память.

Для каждого элемента таблицы настройки к полю сегмента прибавляется сегментный адрес начального сегмента. В результате элемент таблицы указывает на нужное слово в памяти; к этому слову прибавляется сегментный адрес начального сегмента.

Когда таблица настройки адресов обработана, регистрам SS и SP придаются значения, указанные в заголовке, к SS прибавляется сегментный адрес начального сегмента. В ES и DS засылается сегментный адрес начала Префикса Программного сегмента. Управление передается загруженной задаче по адресу, указанному в заголовке (байты 14-17).

Префикса Программного сегмента строится Загрузчиком и содержит системную информацию, необходимую программе. Многие системы используют подобный подход. Так в z/OS такая таблица называется Таблицей Вектора Связей (CVT).

Стандартная связь с другими модулями оформляется в ОС в виде соглашения о связях, которое устанавливает интерфейсы между модулями в среде ОС по управлению и по данным.

Соглашение о связях по управлению определяет вызов модуля и возврат управления в вызывающий модуль. Если осуществляется вызов одной функции из другой, то адрес точки входа и адрес возврата определяют связь между модулями по управлению. Команда вызова использует адрес точки входа для передачи управления, а адрес точки возврата сохраняется и передается вызываемой функции.

Соглашение о связях по данным в случае вызова одной функции из другой осуществляется путем передачи списка параметров. Кроме того, вызываемая функция должна сохранить значения регистров вызывающей функции, а

перед выходом восстановить их. Вызываемая функция возвращает код завершения, который может быть обработан вызывающей функцией.

3.1.1 Соглашение о связях в z/OS MVS

Эта система является результатом разработки целого ряда пакетных систем фирмы IBM. Первые версии этой системы появились еще до изобретения стека, следовательно, не могли использовать этот удобный механизм. Современная система MVS также сохранила все архитектурные черты этих систем.

В системе существуют 16 общих регистров, каждый размером 4 байта (слово) с адресами 0,…15. Каждый из этих регистров может использоваться как для вычислений, так и для работы с адресами. То есть, регистры равноправны, в отличии от архитектуры INTEL, где выделены сегментные регистры, регистр указателя стека и так далее. Но в случае равноправных регистров, программист должен обеспечить их корректное содержимое. Так при входе в каждую секцию необходимо выбрать базовый регистр и установить его значение, сохранить содержимое регистров в области сохранения вызывающей программы, поскольку регистры содержат значения, которые были установлены перед обращением к данной секции. Кроме того, вызываемой программе могут быть переданы параметры.

Таким образом, в соглашениях о связях модулей выделяется управляющая информация: адрес входа, адрес возврата. Также осуществляется связь по данным: передаваемые параметры и область сохранения вызывающей программы. Определенные регистры используются для этих нужд. Назначения регистров показаны в таблице 3.

Таблица 3.

Регистр	Назначение регистра при передаче управления
R1	Адрес списка параметров. Список параметров представляет собой последовательность слов памяти, содержащих адреса передаваемых данных.Конец списка определяется по 1 в левом разряде последнего слова.
R13	Адрес области сохранения вызывающей программы.
R14	Адрес возврата в вызывающую программу.
R15	Адрес точки входа в вызываемую программу в начале работы этой программы. При завершении в этот регистр помещается код возврата (Return Code)

При входе в программу остальные регистры имеют значения, установленные вызывающей программой. Поэтому вызываемая программа должна их сохранить и восстановить при возврате управления вызывающей программе.

Для сохранения содержимого регистров и адреса точки возврата при передаче управления из одного модуля в другой используется область сохранения. В каждом модуле отводится область из 18 четырехбайтных слов (72 байта).

Формат области сохранения показан в таблице 4.

Таблица 4

Слово	Смещение	Назначение
1	0	Используется PL/I и FORTRAN программами
2	4	Адрес области сохранения вызывающей программы (R13)
3	8	Адрес области сохранения программы, вызываемой этой программой
4	12	Адрес возврата в вызывающую программу (R14)
5	16	Адрес точки входа в вызванную программу (R15)
6	20	Содержимое регистра R0
7	24	Содержимое регистра R1
8	28	Содержимое регистра R2
9	32	Содержимое регистра R3
10	36	Содержимое регистра R4
11	40	Содержимое регистра R5
12	44	Содержимое регистра R6
13	48	Содержимое регистра R7
14	52	Содержимое регистра R8
15	56	Содержимое регистра R9
16	60	Содержимое регистра R10
17	64	Содержимое регистра R11
18	68	Содержимое регистра R12

Ниже приводится фрагмент текста на языке IBM ассемблера для программы B, которая вызывается программой A и, в свою очередь вызывает программу C. На рис. 22 показана связь между областями сохранения программ.

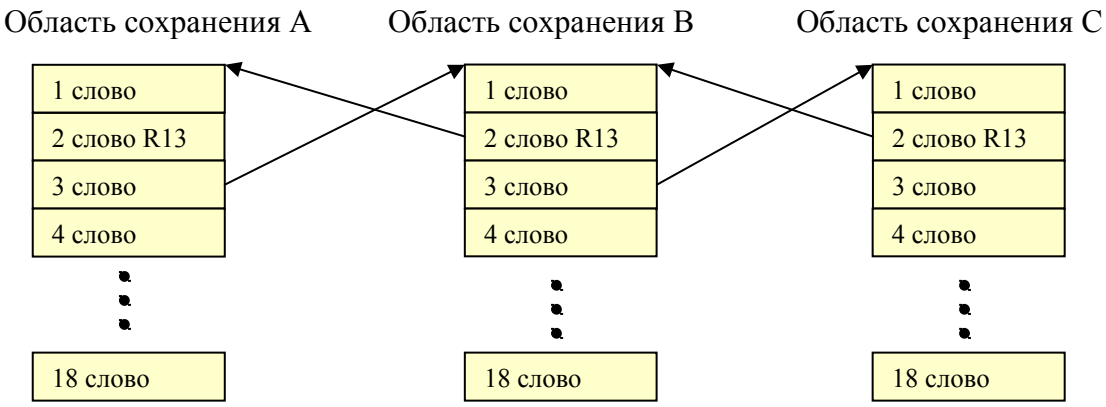


Рис. 22. Связь областей сохранения программы B и программ A и C

Ассемблерный текст программы B содержит только команды, поддерживающие соглашение о связях между модулями. Это стандартные действия и они оформляются в виде макрокоманд.

```
B      START     0          начало программы B
       SAVE      (14,12)         макрокоманда записи в область
*      сохранения программы A содержимого регистров 14, 15,
*      0,...,12
       BASR      12,0       загрузить текущий адрес в Рг 12
       USING     *,12       использовать Рг 12 как базовый,
*      начиная с этого адреса
       ST        13,SV_B+4 записать во второе слово области
*      сохранения программы B адреса области сохранения A
       LA        10,SV_B   загрузить в РГ 10 адрес области
*      сохранения самой программы B
       ST        10,8(13)  запись в третье слово области
*      сохранения программы A адреса области сохранения B
******      Закончено сохранение регистров.

******      Вызов программы C
       LA        13,SV_B   загрузить в Рг 13 адрес области
*      сохранения программы B
       L         15,ENTPNT_C    загрузить в Рг 15 адрес
*      точки входа
       ST        5,PRM5         содержимое регистра 5 в
*      параметры
       BAS       1,GO_C         загрузка в регистр 1 адреса
*      списка параметров и переход к вызову C
PRM5   DS        F              поле содержимого регистра 5
       DC        A(PI-20)  адрес поля PI-20
       DC        XL4'80000000'  конец списка параметров
GO_C   BASR      14,15          перейти по адресу в 15
*      адрес возврата записать в 14
******      Закончен вызов C. Возврат произойдет к следующей
*      за GO_C команде.

******      Выход из программы B
       L         13,SV_B+4      загрузка в 13 адреса области
*      сохранения A
       RETURN          (14,12),RC=0   восстановление регистров
*      программы A, в 15 занести код завершения, выход к
*      адресу в 14
******      Данные программы B
SV_B       DS        18F       область сохранения B
           END
```

3.1.2 Соглашение о связях в системах со стеком

Наличие стека существенно упрощает передачу параметров и связи по управлению между модулями. В архитектуру компьютера вводятся две команды, изменяющие содержимое стека (PUSH и POP) и передающие управление по адресу (CALL и RETurn).

Работа этих команд состоит в следующем. В стек помещается или достается из стека содержимое счетчика адреса команд. В случае команды CALL операндом является адрес перехода к вызываемой подпрограмме. Поэтому, в стек помещается текущее значение счетчика адреса команд (СЧАК), а в СЧАК записывается значение операнда, то есть управление передается по адресу операнда. В случае команды RET верхушка стека заносится в СЧАК, управление возвращается к следующей команде, адрес которой был занесен в стек командой CALL (рис. 23).

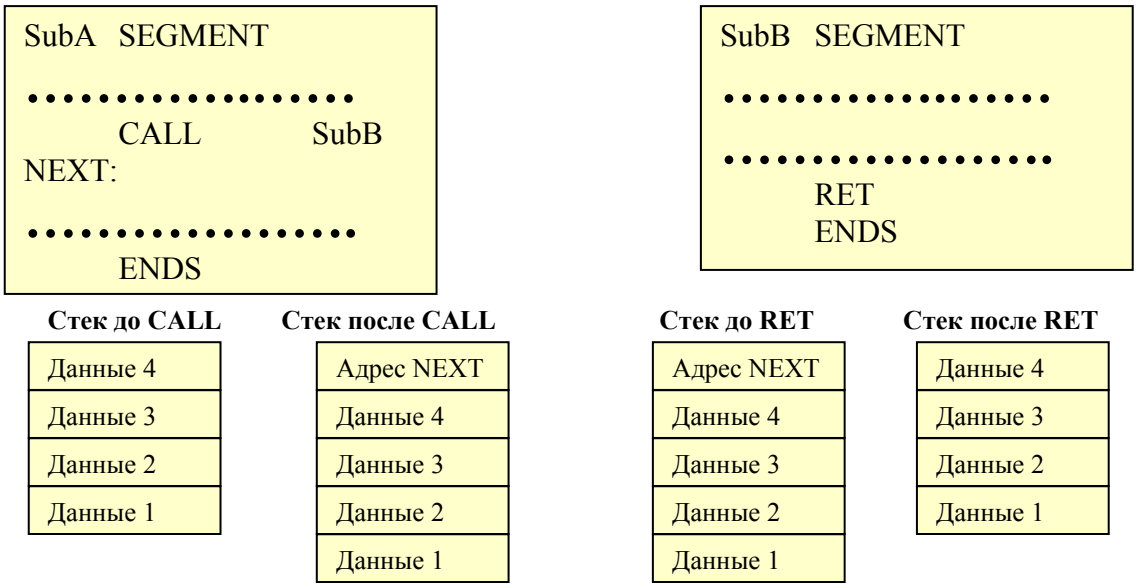


Рис. 23. Состояния стека при вызове программы и выходе из нее

Следующий вопрос связан с передачей параметров в вызываемую программу и получением результатов. Вызывающая программа заносит в стек параметры, а вызываемая программа извлекает их оттуда. Порядок параметров в стеке известен программам. В этом случае используются команды работы со стеком PUSH и POP.

Следующую последовательность команд должна выполнить вызывающая программа. В стек последовательно записываются параметры P1, P2, …,Pn. Затем в стек записывается адрес возврата и осуществляется переход к программе SubB.

Программа SubB, получив управление, сохраняет регистр BP, затем заносит в этот регистр указатель на текущее значение стека, на вершине которого находится сохраненное значение BP, и сохраняет другие регистры, которые будут изменены в подпрограмме. Таким образом в стеке находятся сохраненные регистры, затем идет адрес возврата в программу SubA и параметры. Перед возвратом управления в SubA нужно вынуть из стека занесен-

ные в процессе работы значения, восстановить значения регистров так, чтобы вершине стека оказался адрес возврата в SubA и выполнить команду RET.

Рис.24 иллюстрирует эти действия.

```
SubA PROC                          SubB PROC
                                              PUSH      BP
                                              MOV       BP,SP
                                              PUSH      AX
                                              PUSH      CX
. . . . . . . . . . . .            . . . . . . . . . . . .
          PUSH      P1             P3    EQU       [BP+4]
          PUSH      P2             P2    EQU       [BP+6]
          PUSH      P3             P1    EQU       [BP+8]
          CALL      SubB
NEXT:                              . . . . . . . . . . . .
                                              POP       CX
. . . . . . . . . . . .                       POP       AX
SubA ENDP                                     POP       BP
                                              RET
```

Рис. 24. Передача параметров через стек

Соглашение о связях появилось еще до появления ОС при разработке компиляторов и ассемблеров. Системы программирования поддерживают эти соглашения и при работе с языками высокого уровня программист освобожден от всех этих проблем. Но при программировании на ассемблере соблюдение соглашения о связях обязательно.

Эти соглашения используются и функциями ядра ОС, которые вызываются посредством программного интерфейса. Эти функции также сохраняют значения регистров при получении управления и восстанавливают их по окончанию работы и возвращении управления в программу. Они также используют адрес возврата и код возврата при передаче управления в вызвавшую программу.

Задание для самостоятельной работы!

Задание 6) Изучите соглашения о связях модулей в той ОС, в которой Вы работаете.

3.2 Принцип параметрической настраиваемости

Возможность учета ОС параметров конфигурации ВС называется параметрической настраиваемостью. В настоящее время КС включает в себя широкий спектр устройств различных производителей. ОС должна быть способна распознавать как старые, так новые устройства и обеспечивать их работу. Для этого производители выпускают драйверы устройства для различных ОС. С подключением устройства устанавливается соответствующий драйвер,

который обеспечивает работу с устройством. Управляющая программа учитывает в системных таблицах все подключенные устройства.

В настоящее время устройства выполняются в технологии plug&play, которая предполагает наличия в устройстве информации, которая позволяет ОС опознать тип и модель устройства. Это позволяет автоматизировать процесс установки.

3.3 Принцип функциональной избыточности

Принцип функциональной избыточности состоит во включении в состав ОС различных программных средств, выполняющих одну и ту же функцию, но которые в различных ситуациях работают с различной эффективностью. Дело в том, что к программам предъявляются различные, достаточно противоречивые, требования. Удовлетворить всем этим требованиям с помощью одного варианта программы не удается. Так в управляющей программе существует несколько функций, обеспечивающих различные механизмы синхронизации процессов. В MS DOS существовали три алгоритма управления основной памятью: «Первый подходящий», «Самый подходящий » и «Наименее подходящий».

3.4 Принцип функциональной избирательности

Принцип функциональной избирательности состоит в возможности учета целей, для которых используется система. Воплощение этого принципа позволяет в зависимости от области применения сгенерировать такой вариант системы, который был бы эффективен в данном конкретном применении.

Реализация этого принципа основана на выделении ядра системы, на котором строятся различные версии ОС. Так фирма Microsoft выпустила несколько версий WindowsXP:

Windows XP Professional Edition была разработана для предприятий и предпринимателей и содержит такие функции, как удалённый доступ к рабочему столу компьютера, шифрование файлов (при помощи Encrypting File System), центральное управление правами доступа и поддержка многопроцессорных систем.

Windows XP Home Edition — система для домашнего применения. Выпускается как недорогая «урезанная» версия Professional Edition, но базируется на том же ядре и при помощи некоторых приёмов позволяет провести обновление до почти полноценной версии Professional Edition.

Windows XP Tablet PC Edition базируется на Professional Edition и содержит специальные приложения, оптимизированные для ввода данных стилу-

сом на планшетных персональных компьютерах. Важнейшим свойством является превосходное понимание текстов, написанных от руки и адаптация графического интерфейса к поворотам дисплея. Эта версия продаётся только вместе с соответствующим компьютером.

Windows XP Media Center Edition базируется на Professional Edition и содержит специальные мультимедийные приложения. Компьютер, как правило, оснащён ТВ-картой и пультом дистанционного управления (ПДУ). Важнейшим свойством является возможность подключения к телевизору и управление компьютером через ПДУ благодаря упрощённой системе управления Windows. Эта система содержит также функции для приёма УКВ-радио.

Windows XP Embedded базируется на Professional Edition и предназначена для управления встроенной системой различных устройств: банкоматов, медицинских приборов, кассовых терминалов, игровых автоматов, VoIP-компонентов и т. п.

Windows XP Professional x64 Edition — специальная 64-разрядная версия, разработанная для процессоров с технологией AMD64 Opteron и Athlon 64 от фирмы AMD и процессоров с технологией EM64T от фирмы Intel. Эта система не поддерживает процессоры других производителей, а также не работает с процессором Intel Itanium. Хотя первые 64-разрядные процессоры появились в 2003 году, Windows XP Professional x64 Edition вышла в свет только в апреле 2005 года. Основным достоинством системы является быстрая работа с большими числами (Long Integer и Double Float). Таким образом, эта система очень эффективна, например, при выполнении вычислений, использующих числа с плавающей запятой, необходимых в таких областях, как создание спецэффектов для кинофильмов и трёхмерной анимации, а также разработка технических и научных приложений. Данная система поддерживает смешанный режим, то есть одновременную работу 32- и 64-разрядных приложений, однако для этого все драйверы должны быть в 64-разрядном исполнении. Это означает, что большинство 32-разрядных приложений могут работать и в этой системе. Исключение составляют лишь те приложения, которые сильно зависят от аппаратного обеспечения компьютера, например, антивирусы и дефрагментаторы.

Windows XP Home Edition N и Windows XP Professional Edition N — системы без Windows Media Player и других мультимедиа-приложений. Эти версии созданы под давлением Европейской Антимонопольной Комиссии. При желании пользователь может бесплатно загрузить все недостающие приложения с веб-сайта Microsoft.

Windows XP Starter Edition — сильно функционально ограниченная версия для развивающихся стран и финансово слабых регионов. В этой версии возможна одновременная работа только 3 приложений, и каждое приложение может создать не более 3 окон. В системе полностью отсутствуют сетевые функции, не поддерживается высокая разрешающая способность, а также не допускается использование более 256 мегабайт оперативной памяти или жёсткого диска объёмом более 80 гигабайт.

Такой подход характерен для разработки семейства ОС Linux. Независимая группа разработчиков производит версии ядра, учитывающие требования новых появляющихся процессоров, а на основе этих версий ядра другие разработчики производят варианты ОС, поставляемые потребителям.

3.5 Принцип абстракции и виртуализации

КС существуют и продолжают развиваться благодаря тому, что разработаны по законам иерархии и имеют хорошо определенные интерфейсы, отделяющие друг от друга уровни абстракции. Использование таких интерфейсов облегчает независимую разработку аппаратных и программных подсистем силами разных групп специалистов.

Абстракция скрывает детали реализации нижнего уровня, уменьшая сложность процесса проектирования.

По-существу, ОС представляет собой иерархию уровней абстракции. Так система ввода/вывода позволяет получать доступ к диску на уровне логических блоков. Поэтому следующему уровню (файловой системе), которая обращается к системе ввода/вывода для чтения или записи информации уже нет необходимости знать особенности работы с конкретным типом устройства (например, IDE или SATA). Прикладным программам, которые обращаются к функциям файловой системы, нет необходимости знать, что они работают с диском. Для них информация представляется в виде поименованных совокупностей записей (файлов).

Эти абстракции основываются на хорошо определенных спецификациях интерфейсов. Так спецификация архитектуры системы команд IA-32 позволяет выпускать процессоры фирмам Intel и AMD, на которых работает ПО, разрабатываемое фирмой Microsoft.

Но возникает проблема несовместимости интерфейсов. Разработанные для разных интерфейсов программы не способны работать в других интерфейсах. Например, приложения, распространяемые в двоичных кодах, привязаны к определенной системе команд и к интерфейсам конкретной ОС. Несовместимость интерфейсов особенно актуальна для компьютерных сетей,

где перемещение программ также важно, как и перемещение данных. Виртуализация позволяет преодолеть эту несовместимость интерфейсов.

Виртуализация системы или компонента (например, процессора, памяти или устройства ввода/вывода) на конкретном уровне абстракции отображает его интерфейс и видимые ресурсы на интерфейс и ресурсы реальной системы.

Следовательно, реальная система выступает в роли другой, виртуальной системы или даже нескольких виртуальных систем. В отличие от абстракции, виртуализация не всегда нацелена на упрощение или сокрытие деталей. Так в ОС Microsoft реальный диск можно разбить на два и более логических дисков. Работа с логическим диском будет происходить также как и с реальным диском, то есть упрощения в этом случае не будет. В этом случае не происходит и никакого абстрагирования.

Идея виртуализации применима не только к отдельным подсистемам вроде дисков, но и к машине в целом, то есть строится некоторая виртуальная машина, которая представляет собой слой программного обеспечения, создающий необходимую архитектуру. Таким образом, решается проблема совместимости реальных машин и ресурсных ограничений.

Понятие виртуальной машины тесно связано с понятием компьютерной архитектуры. Под архитектурой какой-либо компьютерной системы или подсистемы понимается описание (спецификация) ее интерфейсов. Так архитектура процессора есть полная и точная спецификация системы команд.

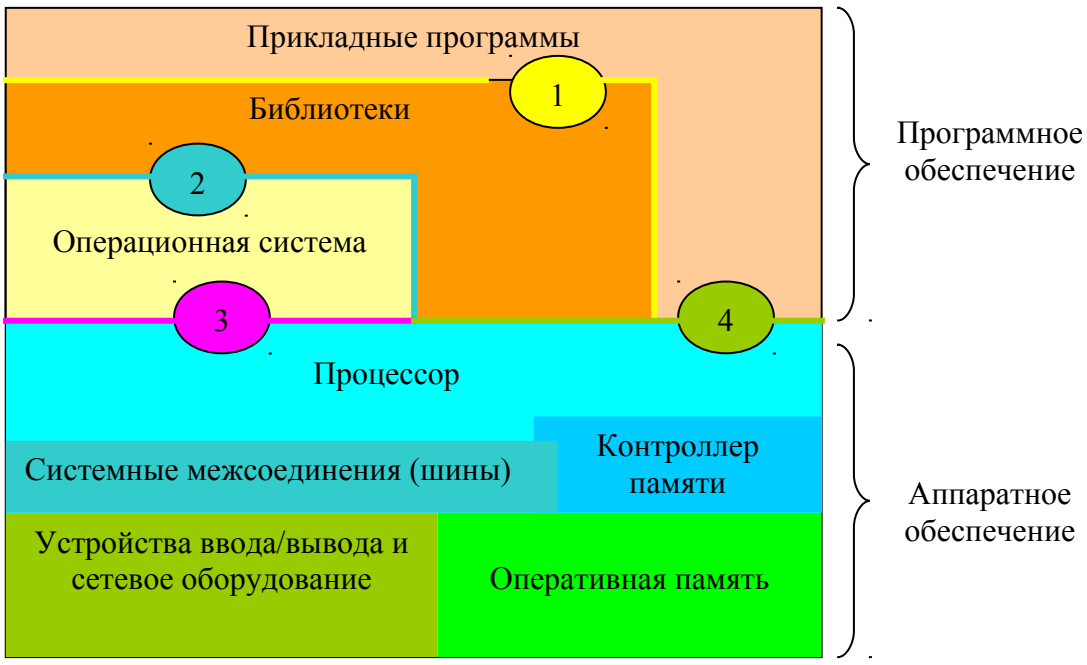


Рис. 25 Архитектура компьютерной системы

Как в оборудовании, так и в программном обеспечении уровни абстракции соответствуют уровням реализации, каждый из которых связан со своим собственным интерфейсом или архитектурой. Таким образом, в компьютерной системе существуют внутренние встроенные системные интерфейсы. На рис. 25 показана архитектура компьютерной системы и наиболее важные интерфейсы.

Интерфейсы 3 и 4 определяют границу между ПО и аппаратурой. Интерфейс 3 представляет собой полное множество команд ISA (instruction set architecture), которое доступно функциям управляющей программы, то есть когда процессор работает в привилегированном (системном) режиме или, как говорят, в режиме супервизора.

Интерфейс 4 использует множество команд, доступных прикладным программам. Процессор, в этом случае, работает в режиме прикладной программы, и выполнение прикладной программой привилегированных команд будет приводить к прерыванию по ошибке.

Интерфейс 2 определяет доступ к функциям управляющей программы. К этим функциям можно обратиться непосредственно из языка Assembler. Для этого необходимо использовать принятое в конкретной ОС соглашение о связях, изложенное в описание функций. Для такого обращения используется синхронное программное прерывание, а в соответствующие регистры записываются передаваемые параметры и номер функции, которую следует выполнить. В качестве примера приводится обращение к функции вывода на экран текстовой информации в MS DOS.

```
mov    DX, offset STRING
mov    AH, 09h
int    21h
```

Первая команда определяет адрес выводимой строки в регистре DX. Вторая определяет номер функции в регистре AH, а третья – осуществляет синхронное прерывание с вектором 21h.

Интерфейс 1 состоит из вызова библиотечных системных процедур в прикладных программах на языках высокого уровня.

Виртуализацию следует рассматривать с точки зрения отдельного процесса и с точки зрения КС. Поэтому выделяют процессную и системную виртуализацию и, соответственно, процессную и системную виртуальные машины.

С точки зрения процесса, обеспечивающего выполнение пользовательской программы, виртуальная машина состоит из выделенного процессу логического адресного пространства, команд пользовательского уровня и реги-

стров, которые позволяют выполнять код этого процесса. Устройства ввода/вывода доступны лишь через управляющую программу ОС, и для процесса есть только один способ взаимодействия с системой ввода/вывода — вызовы системных функций посредством программного интерфейса. Таким образом, процессная виртуальная машина — это виртуальная платформа для выполнения отдельного процесса. Она предназначена исключительно для его поддержки, создается при активизации процесса и прекращает свое существование с его окончанием. На рис. 26 показано как программное обеспечение виртуализации преобразует системные вызовы и команды прикладного уровня в соответствующие вызовы и команды другой аппаратно-программной платформы.

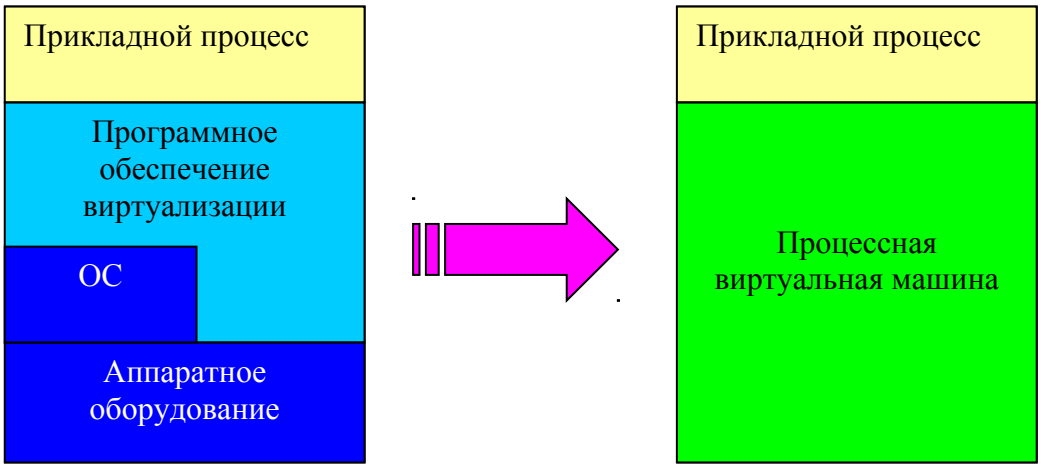


Рис. 26 Процессная виртуальная машина

Системная виртуальная машина предоставляет полнофункциональную, постоянно действующую системную среду для поддержки операционной системы вместе с множеством ее пользовательских процессов; она обеспечивает «гостевой» операционной системе доступ к виртуальным аппаратным средствам, в том числе к процессору и памяти, устройствам ввода/вывода, а иногда — и к графическому интерфейсу (Рис. 27).

Процесс или система, выполняемая на системной виртуальной машине, называется гостем, а базовая платформа, поддерживающая ВМ, — хостом. Программное обеспечение виртуализации, реализующее процессную виртуальную машину, часто для краткости именуют рабочей средой. Программное обеспечение виртуализации системной ВМ обычно называют монитором виртуальных машин (virtual machine monitor, VMM).

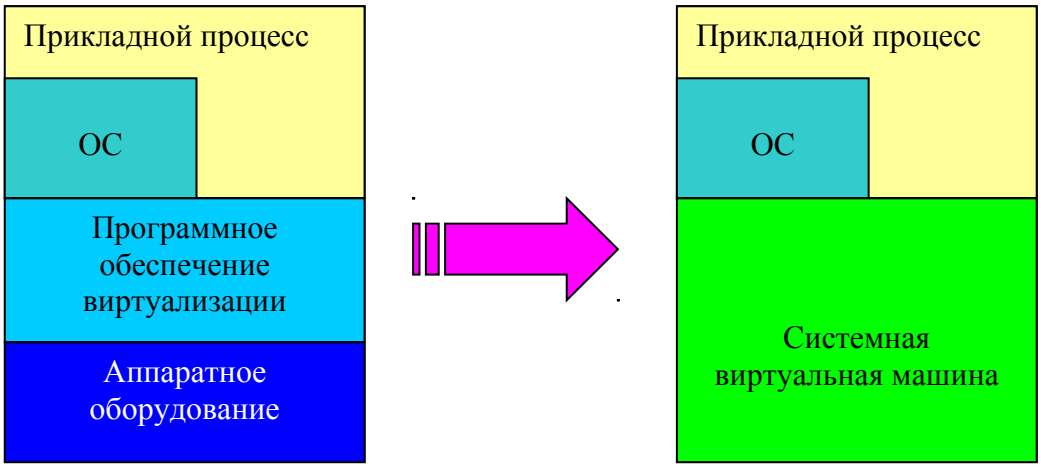


Рис. 27 Системная виртуальная машина

Примером системной виртуализации является ОС VM370 для машин IBM370, которая в настоящее время работает на мэйнфрэймах как z/OS VM. Монитор виртуальных машин, в конечном счете, является управляющей программой, которая управляет аппаратными ресурсами и задачами, которые и являются виртуальными машинами. Модель виртуальной машины показана на рис. 28.

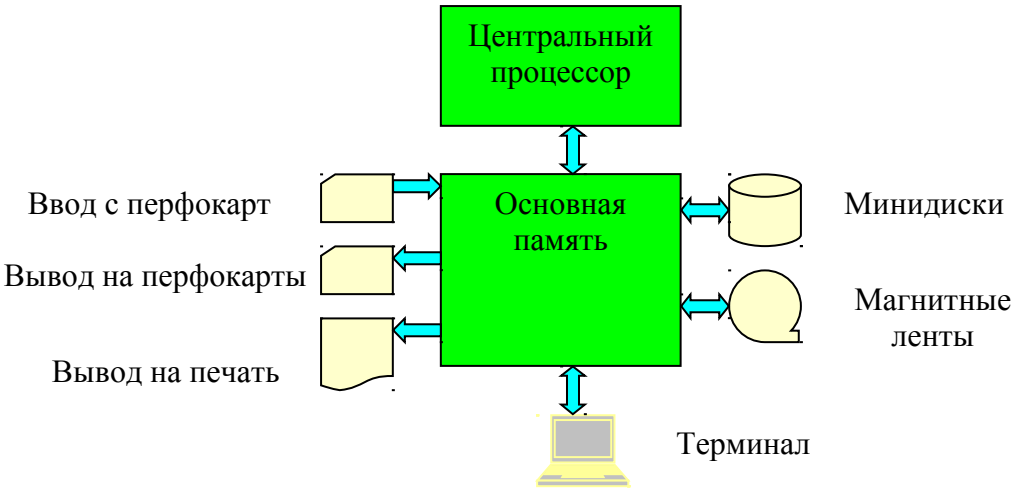


Рис. 28 Системная виртуальная машина

Виртуальная машина имеет виртуальный процессор и виртуальную основную память, а также виртуальные внешние устройства. Все эти устройства моделируются монитором виртуальных машин на реальных аппаратных средствах.

Виртуальный центральный процессор моделируется предоставлениями квантов времени реального ЦП.

Основная память виртуальной машины моделируется страницами памяти, выделяемой монитором виртуальных машин каждой виртуальной машине.

Внешние устройства ввода и вывода с перфокарт моделируются файлами с записями фиксированной длины 80 байт. Это удобное средство обмена между виртуальными машинами. Пользователь одной виртуальной машины может вывести на устройство вывода с перфокарт файл и указать имя виртуальной машины другого пользователя. Этот файл поступит на устройство ввода с перфокарт этой машины и другой пользователь может прочитать этот файл. Для такой передачи необходимо знать лишь имя виртуальной машины и не нужно знать пароль другой машины.

Тема 4. КОМАНДНЫЙ И ПРОГРАММНЫЙ ИНТЕРФЕЙСЫ ОС

4.1 Командный интерфейс ОС

Командный интерфейс обеспечивает взаимодействие человека и компьютера. Для того, чтобы осуществлять такую связь в ОС сразу после загрузки стартуют определенные системные задачи, которые воспринимают команды, посылаемые с терминала, и выполняют их. В пакетных и диалоговых системах это выполняется различным способом.

4.1.1 *Командный интерфейс в мультипрограммной пакетной ОС*

В пакетной мультипрограммной ОС z/OS MVS такими системными задачами являются: JES (Job Entry System), которая обрабатывает поток заданий, TSO (Time Sharing Option), которая поддерживает диалоговый режим работы с пользователями, и ISPF/PDF (Interactive System Productivity Facility/ Programm Development Facility), которая позволяет набирать и редактировать тексты программ, в том числе и задания на JCL, и посылать задания во входной поток JES, а также выполнять много других функций. Доступ к системе осуществляется с персонального компьютера, на котором установлена какая-либо программа, предназначенная для связи с z/OS по протоколу TelNet.

Таким образом, средством выполнения приложений является язык управления заданиями (JCL). Пользователь КС описывает последовательность задач, которые должны быть выполнены, в виде задания. Каждой задаче соответствует шаг задания. В каждом шаге описывается программа, которая выполняется в этой задаче и данные, используемые этой программой. Планировщик заданий обрабатывает и исполняет все задания, поступающие в систему. JCL это достаточно сложный, стенографический язык, весьма неудобный в использовании. Также как и пакетный режим обработки неудобен для отладки программ и взаимодействия человека с компьютером. Однако, если требуется проводить многократную периодическую обработку в течение продолжительного времени, то такой способ имеет много преимуществ.

Рассмотрим пример задания для ассемблирования, компоновки и выполнения ассемблерной программы.

Текст на JCL форматирован. Первые две позиции определяют тип строки. Если символы // в начале строки, то это оператор языка. /* - символы конца данных. //* - строка комментарий. Пробел является разделителем, поэтому хвост строки после пробела воспринимается как комментарий.

```
//USUAL JOB A2317P,'MAE BIRDSALL'
//ASM EXEC PGM=IEV90,REGION=256K, EXECUTES ASSEMBLER
```



```
// PARM=(OBJECT,NODECK,'LINECOUNT=50')
//SYSPRINT DD SYSOUT=*,DCB=BLKSIZE=3509 PRINT THE ASSEMBLY LISTING
//SYSPUNCH DD SYSOUT=B PUNCH THE ASSEMBLY LISTING
//SYSLIB DD DSNAME=SYS1.MACLIB,DISP=SHR THE MACRO LIBRARY
//SYSUT1 DD DSNAME=&&SYSUT1,UNIT=SYSDA, A WORK DATA SET
//            SPACE=(CYL,(10,1))
//SYSLIN DD DSNAME=&&OBJECT,UNIT=SYSDA, THE OUTPUT OBJECT MODULE
//          SPACE=(TRK,(10,2)),DCB=BLKSIZE=3120,DISP=(,PASS)
//SYSIN DD * IN-STREAM SOURCE CODE
.
code
.
/*
//LKED EXEC PGM=HEWL, EXECUTES LINKAGE EDITOR
// PARM='XREF,LIST,LET',COND=(8,LE,ASM)
//SYSPRINT DD SYSOUT=* LINKEDIT MAP PRINTOUT
//SYSLIN DD DSNAME=&&OBJECT,DISP=(OLD,DELETE) INPUT OBJECT MODULE
//SYSUT1 DD DSNAME=&&SYSUT1,UNIT=SYSDA, A WORK DATA SET
// SPACE=(CYL,(10,1))
//SYSLMOD DD DSNAME=&&LOADMOD,UNIT=SYSDA, THE OUTPUT LOAD MODULE
// DISP=(MOD,PASS),SPACE=(1024,(50,20,1))
//GO EXEC PGM=*.LKED.SYSLMOD,TIME=(,30), EXECUTES THE PROGRAM
// COND=((8,LE,ASM),(8,LE,LKED))
//SYSUDUMP DD SYSOUT=* IF FAILS, DUMP LISTING
//SYSPRINT DD SYSOUT=*, OUTPUT LISTING
// DCB=(RECFM=FBA,LRECL=121)
//OUTPUT DD SYSOUT=A, PROGRAM DATA OUTPUT
// DCB=(LRECL=100,BLKSIZE=3000,RECFM=FBA)
//INPUT DD * PROGRAM DATA INPUT
  data
/*
//
```

4.1.2 Командный интерфейс в диалоговых ОС

В диалоговых ОС вместо JCL используется командный язык, который обрабатывается процессором командного языка. Процессор командного языка является системной задачей, которая создается сразу после загрузки ОС. Он воспринимает команды, вводимые пользователем с терминала, и исполняет их.

Такой подход принципиально отличен от подхода, принятого в пакетной системе. Это отличие состоит в том, что при написании задания в пакетной системе предварительно задаются все сведения о программе, о наборах данных, используемых программой. Поэтому пакетная система использует принцип предварительного распределения ресурсов перед выполнением задачи шага задания. Это влияет на организацию системы и алгоритмы управления, реализованные в системе.

В диалоговых системах все действия по предоставлению ресурсов стартовавшей задаче производятся во время ее выполнения, поскольку вся информация содержится в коде запущенной задачи.

Командный язык (CLI – Command Line Interface) состоит из команд, которые вводятся с клавиатуры. Во-первых, следует отметить определенные

клавиши или комбинации клавиш. Такие, например, как Ctrl-C, Ctrl-Break, Ctrl-Alt-Del. Эти комбинации воспринимаются процессором командного языка и приводят к определенным действиям.

Другой источник команд - это команды, вводимые пользователем в командной строке. Часть этих команд выполняется самим командным процессором. Другие связаны с выполнением утилит. С этой точки зрения все исполнимые файлы могут рассматриваться как команды, поскольку командный процессор запускает их стандартным образом.

Формат команд в основном следующий:

com1 par1,par2,...,parn

com1 – это либо имя внутренней команды, либо путь к программе в каталоге. Внутренние команды выполняются самим командным процессором и являются достаточно простыми. Если требуется выполнить более сложные действия, например, копирование файлов, то командный процессор выполняет соответствующую утилиту.

Однако, в ряде случаев, удобно выполнять некоторую последовательность команд. Поэтому в диалоговых ОС возможность такого «пакетного» исполнения была сохранена в виде командных файлов (batch файлов). Язык командных файлов состоит из директив и команд. Директивы указывают командному процессору порядок выполнения команд и управляют выполнением командного файла, а команды непосредственно исполняются командным процессором.

Перечень команд, который можно получить по команде HELP, запустив cmd.exe в Windows:

```
    ASSOC     Вывод либо изменение сопоставлений по расширениям имен файлов.
    AT        Выполнение команд и запуск программ по расписанию.
    ATTRIB    Отображение и изменение атрибутов файлов.
    BREAK     Включение/выключение режима обработки комбинации клавиш CTRL+C.
    CACLS      Отображение/редактирование списков управления доступом (ACL) к
файлам.
    CALL      Вызов одного пакетного файла из другого.
    CD        Вывод имени либо смена текущей папки.
    CHCP      Вывод либо установка активной кодовой страницы.
    CHDIR     Вывод имени либо смена текущей папки.
    CHKDSK    Проверка диска и вывод статистики.
    CHKNTFS   Отображение или изменение выполнения проверки диска во время за-
грузки.
    CLS       Очистка экрана.
    CMD       Запуск еще одного интерпретатора командных строк Windows.
    COLOR     Установка цвета текста и фона, используемых по умолчанию.
    COMP      Сравнение содержимого двух файлов или двух наборов файлов.
    COMPACT   Отображение/изменение сжатия файлов в разделах NTFS.
    CONVERT   Преобразование дисковых томов FAT в NTFS. Нельзя выполнить
              преобразование текущего активного диска.
    COPY      Копирование одного или нескольких файлов в другое место.
    DATE      Вывод либо установка текущей даты.
    DEL       Удаление одного или нескольких файлов.
```



```
    DIR      Вывод списка файлов и подпапок из указанной папки.
    DISKCOMP Сравнение содержимого двух гибких дисков.
    DISKCOPY Копирование содержимого одного гибкого диска на другой.
    DOSKEY   Редактирование и повторный вызов командных строк; создание ма-
кросов.
    ECHO      Вывод сообщений и переключение режима отображения команд на
экране.
    ENDLOCAL Конец локальных изменений среды для пакетного файла.
    ERASE    Удаление одного или нескольких файлов.
    EXIT      Завершение работы программы CMD.EXE (интерпретатора командных
строк).
    FC        Сравнение двух файлов или двух наборов файлов и вывод различий
между ними.
    FIND     Поиск текстовой строки в одном или нескольких файлах.
    FINDSTR  Поиск строк в файлах.
    FOR      Запуск указанной команды для каждого из файлов в наборе.
    FORMAT   Форматирование диска для работы с Windows.
    FTYPE    Вывод либо изменение типов файлов, используемых при сопоставле-
нии по расширениям имен файлов.
    GOTO     Передача управления в отмеченную строку пакетного файла.
    GRAFTABL Позволяет Windows отображать расширенный набор символов в графи-
ческом режиме.
    HELP     Выводит справочную информацию о командах Windows.
    IF       Оператор условного выполнения команд в пакетном файле.
    LABEL    Создание, изменение и удаление меток тома для дисков.
    MD       Создание папки.
    MKDIR    Создание папки.
    MODE     Конфигурирование системных устройств.
    MORE     Последовательный вывод данных по частям размером в один экран.
    MOVE      Перемещение одного или нескольких файлов из одной папки в дру-
гую.
    PATH     Вывод либо установка пути поиска исполняемых файлов.
    PAUSE    Приостановка выполнения пакетного файла и вывод сообщения.
    POPD     Восстановление предыдущего значения текущей активной папки,
             сохраненного с помощью команды PUSHD.
    PRINT    Вывод на печать содержимого текстовых файлов.
    PROMPT   Изменение приглашения в командной строке Windows.
    PUSHD     Сохранение значения текущей активной папки и переход к другой
папке.
    RD       Удаление папки.
    RECOVER  Восстановление читаемой информации с плохого или поврежденного
диска.
    REM      Помещение комментариев в пакетные файлы и файл CONFIG.SYS.
    REN      Переименование файлов и папок.
    RENAME   Переименование файлов и папок.
    REPLACE  Замещение файлов.
    RMDIR    Удаление папки.
    SET      Вывод, установка и удаление переменных среды Windows.
    SETLOCAL Начало локальных изменений среды для пакетного файла.
    SHIFT     Изменение содержимого (сдвиг) подставляемых параметров для па-
кетного файла.
    SORT     Сортировка ввода.
    START    Запуск программы или команды в отдельном окне.
    SUBST    Сопоставляет заданному пути имя диска.
    TIME     Вывод и установка системного времени.
    TITLE    Назначение заголовка окна для текущего сеанса интерпретатора
             командных строк CMD.EXE.
    TREE      Графическое отображение структуры папок заданного диска или за-
данной папки.
    TYPE     Вывод на экран содержимого текстовых файлов.
    VER      Вывод сведений о версии Windows.
    VERIFY   Установка режима проверки правильности записи файлов на диск.
    VOL      Вывод метки и серийного номера тома для диска.
    XCOPY    Копирование файлов и дерева папок.
```


С развитием периферийных устройств командный интерфейс стал существенно более дружественным и многообразным. Его основу составил Графический интерфейс пользователя (GUI – graphic user interface).

4.2 Программный интерфейс ОС

Программный интерфейс (API – application program interface) предназначен для программистов и служит для использования в разрабатываемых программах функций ОС. Для реализации API используется синхронное (программное) прерывание. Вызов из программы функции ядра ОС состоит в выполнении прерывания. Предварительно в регистры загружаются параметры и в определенный регистр код функции. При выполнении прерывания управление передается ядру ОС. Функция ядра, обрабатывающая прерывание, анализирует код функции и передает управление функции ОС, определяемой этим кодом (рис. 29).

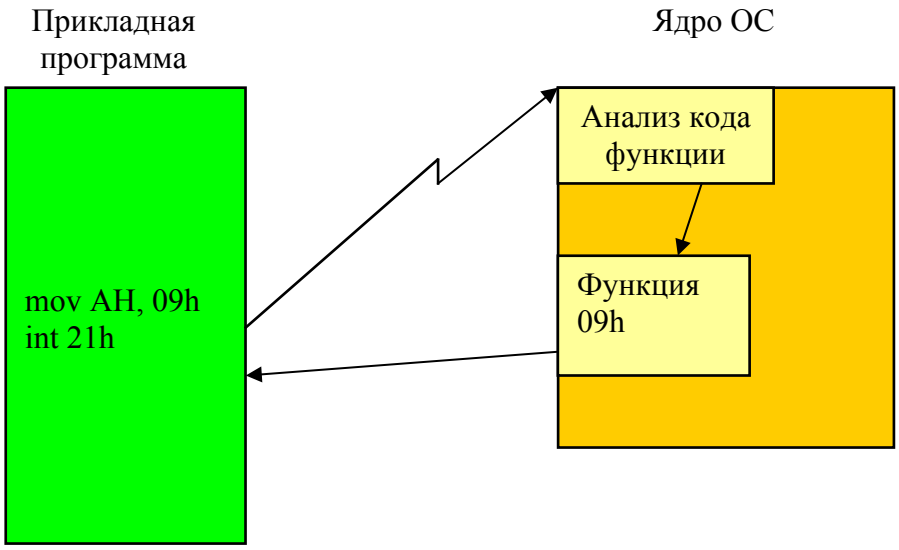


Рис. 29. Реализация программного интерфейса

Тема 5. ОРГАНИЗАЦИЯ И ТИПЫ ПРОГРАММНЫХ МОДУЛЕЙ

5.1 Загрузка исполнимых модулей в основную память

Исполнимые (загрузочные) модули создаются системой программирования в результате компиляции из исходного текста в код компьютера. В процессе исполнения копия загрузочного модуля переносится загрузчиком из файла на диске в ОП. Загрузчик представляет собой функцию ядра, отвечающую за загрузку и настройку адресов исполнимого модуля на адреса раздела ОП, в которую помещается этот модуль.

Процесс загрузки исполнимого модуля в основную память связан с преобразованием адресов. Изготовленный загрузочный модуль хранится в файле на диске. В начале этого файла располагается таблица настройки – Relocation Table (См. раздел 3.1. Принцип модульности). Затем идет непосредственно исполнимый модуль. Этот модуль может содержать адреса, значения которых определяются только при загрузке и размещении модуля в основной памяти. Это прямые адреса. Возможность появления этих адресов определяется системой адресации компьютера. Примеры таких адресов приведены ниже.

1) mov AX,seg DATA ; в регистр AX загружается адрес сегмента DATA
2) call far ptr SUB_FAR ; вызов процедуры из другого сегмента
3) jmp far ptr LBL_FAR ; переход к метке в другом сегменте

В первом случае адрес сегмента не известен до загрузки этого сегмента в память и определяется Загрузчиком. Следовательно, только этот модуль управляющей программы может откорректировать эти команды по месту в памяти. В оставшихся случаях также неизвестны до загрузки в память адреса сегментов, в которых находятся подпрограмма и метка.

Для обеспечения настройки адресов Загрузчику необходима информация о местоположении в файле загрузочного модуля полей адресов, требующих настройки на адреса основной памяти и значение действительных адресов, которые будут помещены в эти поля. Информация о типе адресов и их местоположении в коде хранится в таблице настройки. Загрузчик читает таблицу настройки и код и по таблице, в соответствии с расположением программы в памяти, заносит действительные адреса.

Другой вопрос, который связан с помещением в основную память загрузочного модуля, относится к структуре исполнимого модуля в файле. Дело в том, что код модуля состоит из сегментов. Но в различных системах деление

на сегменты производится различным образом. В системе z/OS MVS программа состоит из секций, куда помещаются и код и данные. Каждая секция является сегментом, в котором могут быть процедуры и данные. Идея разделить код и данные появилась в UNIX. В подобных системах строятся сегмент кода, сегмент данных и динамический сегмент, содержащий стек и кучу.

При помещении в файл загрузочного модуля системы программирования также поступают разным образом. Наиболее простой способ состоит в создании образа этого модуля в основной памяти. В этом случае в файле все сегменты размещаются с соответствующих адресов, относительно начала. В файле отводится место для стека. Такой файл имеет избыточное пространство и занимает больше места. Но при помещении модуля в основную память нужно просто переписать его из файла и настроить адреса.

Другая реализация старается сократить этот объем за счет того, что динамический сегмент не хранится на диске. Он строится Загрузчиком при загрузке и настройке исполнимого модуля.

После размещения и настройки исполнимого модуля, Загрузчик настраивает значения регистров. Это возможно в системах, в которых выделяются сегментные регистры, регистр указателя стека и счетчик адреса команд. Это характерно для архитектуры INTEL. Загрузчик заносит адрес сегмента кода в сегментный регистр CS, адрес сегмента стека в сегментный регистр SS, устанавливает регистр указателя стека SP, в регистры DS и ES помещается адрес системной таблицы, в которой находятся адреса и данные на системные ресурсы. И завершается работа Загрузчика помещением в СЧАК адреса точки входа. Таким образом, управление передается загруженной программе.

В системе z/OS регистры равноправны. Любой из 16 общих регистров может быть как базовым, так и использоваться для данных. В такой ситуации Загрузчик не может определять назначения регистров, эта работа выполняется программистом.

5.2 Преобразование адресов программы в компьютерной системе

Таким образом, в процессе компиляции или ассемблирования модуля, в процессе компоновки объектных модулей, при загрузке исполнимого модуля в ОП компьютера, а также при выполнении загрузочного модуля процессором происходит преобразование адресов. На рис. 30 показаны этапы преобразования адресов.

При написании программы, используются символические имена для переменных. Эти имена преобразуются системой программирования в про-

граммные адреса. Структура программного адреса зависит от системы адресации компьютера и организации основной памяти.

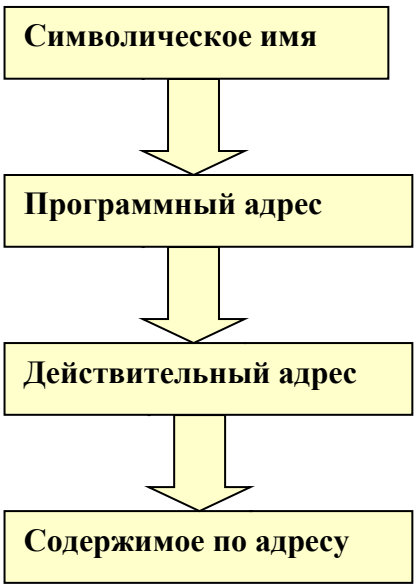


Рис. 30

Программный адрес преобразуется системой адресации компьютера или Загрузчиком в действительный адрес.

По действительному адресу система памяти компьютера читает или записывает данные.

Поскольку, преобразование символического имени в программный адрес осуществляется системой программирования, то эти вопросы рассматриваются в курсе, посвященном конструированию компиляторов. Взятие содержимого по действительному адресу выполняет аппаратура компьютера и эти вопросы рассматриваются в курсах по организации и функционированию компьютерных систем.

В дисциплине «Операционные системы» следует рассмотреть преобразование программного адреса в действительный адрес, поскольку это тесно связано с организацией управляющей программы. Прежде всего, это влияет на привязывание адресного пространства загрузочного модуля к выделенной физической памяти. С этой точки зрения методы связывания бывают статические и динамические.

Статические методы связывания программного адреса

Статическое связывание осуществляется до выполнения загрузочного модуля после загрузки его в основную память и состоит в редактировании адресов, которые не могут быть определены системой программирования и компоновщиком. Эти адреса определяются загрузчиком в момент перемещения копии программы с диска в память.

Примеры таких адресов приведены в 5.1. Используя таблицу настройки, Загрузчик знает из таблицы настройки, где располагаются поля адресов в коде программы и какие значения следует туда поместить.

В этом случае загрузочный модуль содержит в своем начале таблицу настройки. В таблице 2 (см. 3.1. Принцип модульности) показана структура такой таблицы.

Динамические методы связывания программного адреса

Динамическое связывание осуществляется в момент выполнения программы и состоит в вычислении действительного адреса аппаратными средствами компьютера. Это зависит от способа вычисления адреса в команде.

Наиболее распространенным является способ представления адреса с базированием. В этом случае отводятся два или более полей в адресной части команды. Одно поле занимает смещение. Это относительный адрес относительно базового адреса. Этот относительный адрес определяется системой программирования и не требует коррекции при загрузке в основную память. Другие поля занимают адреса базового и, возможно, индексного регистра. Адреса регистров также определяются во время компиляции при распределении регистров во время генерации кода в компиляторе. Значения регистров определяются при загрузке модуля в память. Для определения действительного адреса происходит суммирование смещения с содержимым регистров (Рис. 31).

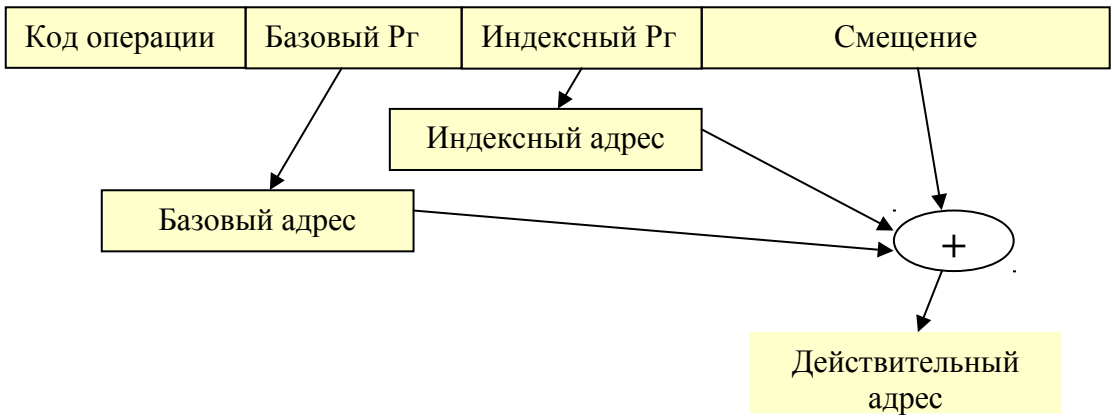


Рис. 31. Вычисление действительного адреса по смещению, базовому адресу и индексу

Динамическое связывание также обеспечивается страничной, сегментной и сегментно-страничной организацией памяти. Следует отметить, что эти методы адресации не отменяют использования базовых и индексных регистров. Но в этом случае вычисленный в результате суммирования смещения и значений базового и индексного регистров адрес является программным адресом, который и служит исходным для вычисления действительного адреса.

В случае страничной организации памяти программный адрес разделяется на два поля. Первое рассматривается как номер страницы, а второе как смещение на странице. Размер страницы фиксирован. Наиболее популярный размер страницы 4096 байт. В этом случае на смещение нужно отвести 12 разрядов. Оставшиеся разряды программного адреса определяют номер стра-

ницы. На рис. 32 показан способ вычисления действительного адреса по программному адресу.

В страничной системе при загрузке задачи в основную память под ее адресное пространство отводятся свободные страницы и строится таблица страниц. Элементы таблицы страниц заполняются информацией о том, находится ли страница в основной памяти или она находится на диске. Если страница загружена в основную память, то в соответствующем элементе таблицы хранится базовый адрес физической страницы. Этот адрес суммируется со смещением, указанным в поле программного адреса. Таким образом, вычисляется действительный адрес.

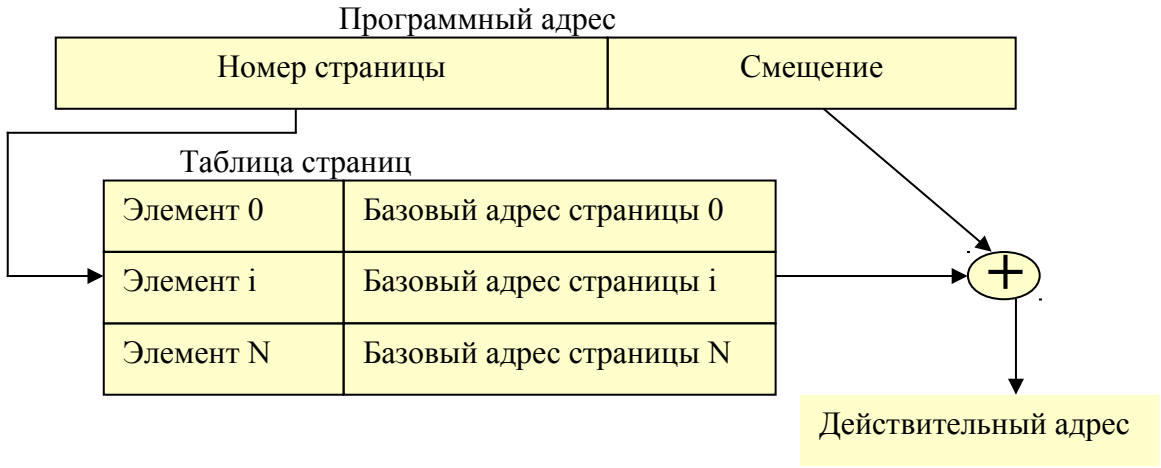


Рис. 32. Вычисление действительного адреса по программному адресу

Это выполняется системой адресации компьютера при разборе команды. По номеру страницы определяется элемент таблицы страниц, который загружается на управляющий регистр. Если установлен бит отсутствия страницы в основной памяти, то возникает страничное прерывание. Если страница находится в ОП, то вычисляется действительный адрес.

5.3 Типы загрузочных модулей

В ходе эволюции ОС использовались различные типы загрузочных модулей. Вначале это было связано с ограниченным объемом ОП. Впоследствии с появлением ОП большого объема в UNIX и UNIX-подобных ОС стал использоваться один тип - модуль простой структуры. В z/OS MVS используется и модуль динамической структуры. Динамическая компоновка удобна в ряде случаев и в Windows реализованы DLL.

5.3.1 Загрузочный модуль простой структуры

Загрузочный модуль простой структуры загружается в основную память целиком, то есть все логическое адресное пространство перемещается в физическую память.

Загрузочный модуль простой структуры создается из совокупности объектных модулей, полученных в результате компиляции, и библиотечных модулей. Этот процесс называется связыванием объектных модулей, а при его выполнении происходит разрешение внешних ссылок.

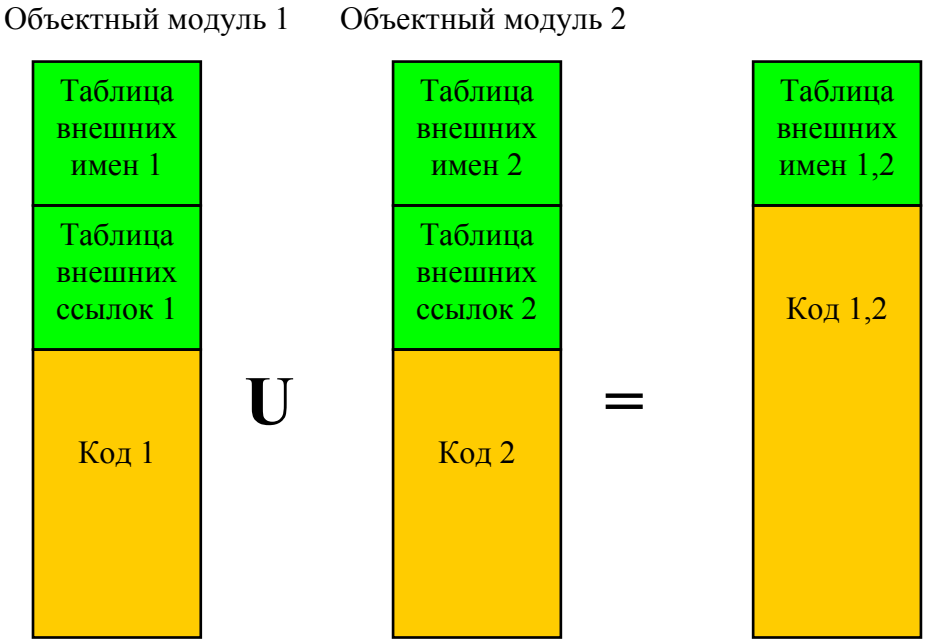


Рис.33 . Связывание двух объектных модулей

Пусть имеется m объектных модулей (рис. 33). Каждый модуль начинается с адреса 0 и в начале содержит таблицу внешних имен и таблицу внешних ссылок. Связывание модулей начинается с первых двух объектных модулей. К коду первого модуля приписывается с некоторого адреса A код второго модуля. Все адресно-зависимые элементы второго модуля увеличиваются на константу A. Таблицы внешних имен двух модулей объединяются, причем адреса внешних символов второго модуля также корректируются. По полученной таблице внешних имен и таблицам внешних ссылок осуществляется их разрешение. Для этого по имени внешней ссылки в таблице внешних имен ищется адрес, соответствующий этому имени. Если он найден, то этот адрес подставляется на место ссылки. Таким образом, компоновщик старается разрешить все внешние ссылки.

К полученному на предыдущем этапе коду добавляется код следующего объектного модуля, в котором корректируются адресно-зависимые элементы, корректируются адреса в таблицах внешних имен и внешних ссылок, осуще-

ствляется разрешение внешних ссылок. Таким образом, обрабатываются все объектные модули. Если все внешние ссылки разрешены, то есть для каждой внешней ссылки найден адрес в таблице внешних имен, то построение загрузочного модуля заканчивается успешно. В противном случае выдается сообщение: «Unresolved external reference». Это означает, что нет каких-то объектных модулей, либо не подключены какие-то библиотеки.

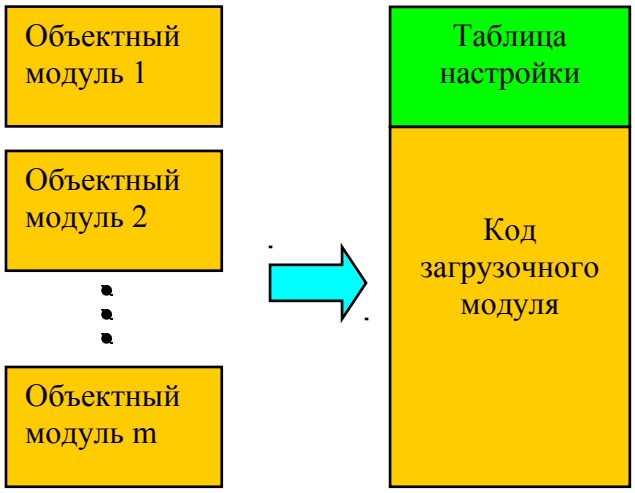


Рис.34 . Построение загрузочного модуля

Построенный загрузочный модуль (рис. 34) содержит код и таблицу адресно-зависимых элементов, необходимую для настройки модуля на адреса основной памяти при загрузке.

5.3.2 Загрузочный модуль оверлейной структуры

Модуль оверлейной структуры использовался в случае, когда объем основной памяти был недостаточен для размещения всего логического адресного пространства. В этом случае загрузочный модуль состоит из нескольких сегментов, которые по мере необходимости загружаются в основную память.

Среди этих сегментов выделяется корневой сегмент, который постоянно находится в основной памяти и содержит начальную точку входа. Остальные сегменты загружаются по мере передачи управления их коду. Причем при загрузке они могут перекрывать адресные пространства других сегментов в памяти, которые в этот момент не активны, то есть в эти сегменты не будет передачи управления. Для организации такого исполнения необходима таблица, которая отражает возможности взаимного перекрытия сегментов. Схема иерархии сегментов для тривиального случая показана на рис. 35.

Сегмент A является корневым сегментом и постоянно находится в памяти. Из кода сегмента A существует передача управления в код сегмента B, либо в код сегмента C. Соответственно, из сегмента B управление может быть передано в D или в E. Также из сегмента C управление передается либо

в F, либо в G. Таким образом, перекрываться могут пары сегментов B и C, D и E, F и G.

Рассмотрим возможную схему выполнения такого модуля, когда из сегмента A управление передается в B. Затем из сегмента B управление передается в D. Из сегмента D осуществляется возврат управления в B и управление передается в E. Сегмент E перекрывает в памяти сегмент D и загружается на его место.

Затем управление возвращается в сегмент B, а затем из B в A. Таким образом, память, занимаемая B и D, свободна и при обращении из сегмента A к сегменту C он будет загружен на место B. При вызове из C сегментов F и G они также будут перекрываться.

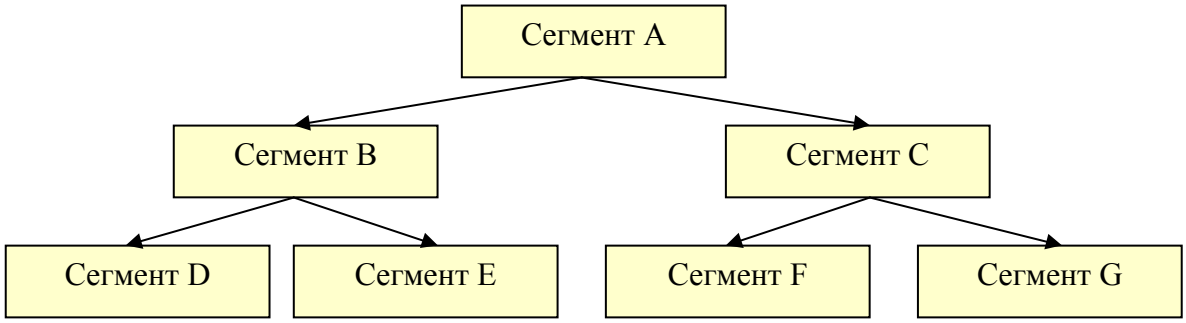


Рис. 35. Схема иерархии сегментов.

На рис. 36 показаны изменения карты памяти при выполнении оверлейного модуля.

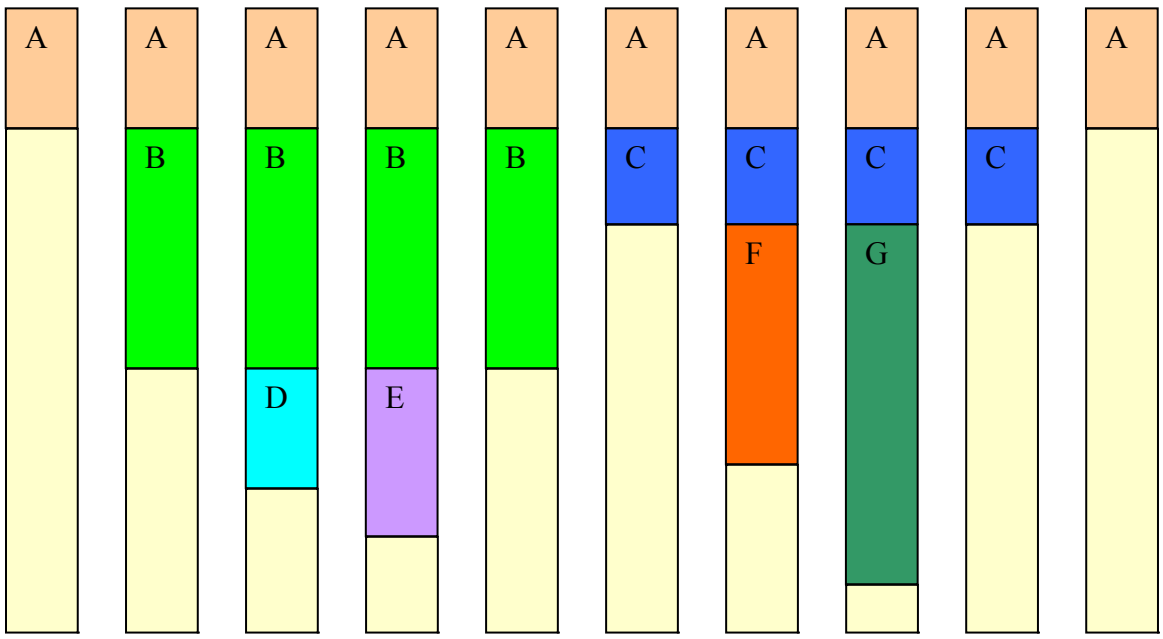


Рис. 36. Изменение карты памяти оверлейного модуля.

Для построения загрузочного модуля оверлейной структуры необходимо построить сегменты и спланировать структуру модуля. Затем эта структура

описывается управляющими предложениями Компоновщика (Редактора связей – Linker).

Для построения сегментов необходимо, чтобы система программирования изготавливала объектные модули для программных секций. Программная секция является наименьшей независимой перемещаемой частью программы. Программными секциями в языках высокого уровня являются функции и подпрограммы. Так в языке ФОРТРАН каждая подпрограмма или функция транслировалась в отдельный объектный модуль, даже если они располагались в одном файле.

Иное дело системы программирования для языка Си. Поскольку язык Си тесно связан с ОС UNIX, а в этой системе от оверлеев отказались, то каждый файл .c или .cpp транслируется в объектный модуль, а не отдельные функции, входящие в него.

Таким образом, множество программных секций представлено множеством объектных модулей, полученных компилятором. Для этих программных секций строится схема иерархии, которая отражает, из какой секции какие другие секции вызываются. Это и является основой для построения модуля оверлейной структуры. В ходе планирования необходимо получить сегменты, включающие в себя программные секции, и порядок их загрузки в основную память.

Планирование оверлейной структуры зависит от связей между секциями программы. Если одна программная секция имеет ссылки к другой программной секции, то эти секции зависимы. Схема иерархии программы отражает зависимость секций. Схему иерархии можно представить в виде графа, в котором вершинами являются программные секции, а дуги указывают обращения из одной секции к другой. Такой граф имеет вид дерева. На рис. 37 показан пример такого графа.

Секции необходимо объединить в сегменты. Компоновщик накладывает следующее условие: каждая секция может находиться только в одном сегменте. Это связано с экономией памяти. Сама идея оверлейного модуля предназначена для экономии памяти, поэтому расточительно загружать несколько копий программной секции в память. Это ограничение должно учитываться при планировании. То есть каждая секция может входить только в один сегмент.

Кроме того, должно выполняться еще условие при построении дерева сегментов, которое состоит в том, что секции, к которым происходит обращение из текущего сегмента, должны находиться в сегментах уже находящихся

в памяти или в сегментах зависящих от этого сегмента. Для показанного на рис. 37 дерева секций можно построить следующее дерево сегментов (рис.38).

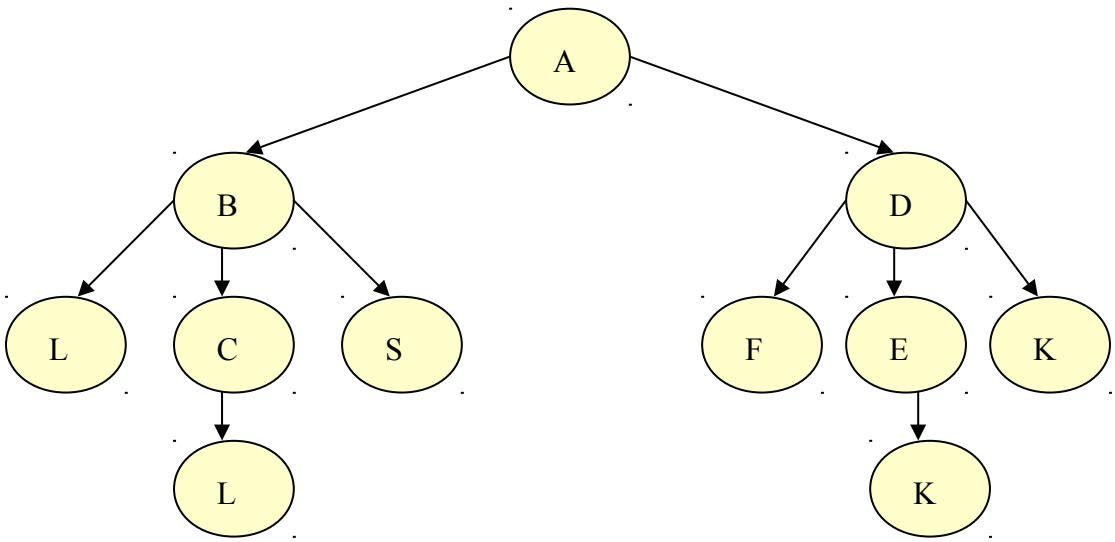


Рис. 37. Схема иерархии программных секций.

В дереве секций (рис.37) существует обращение к секции L из секции C. В дереве сегментов секция L будет находиться в памяти в момент передачи управления секции C. То же можно сказать и о секции K. Таким образом эта оверлейная структура построена правильно. Однако, дерево секций может иметь такую структуру, что описанным способом невозможно построить перекрытия, то есть в результате планирования все секции объединятся в один корневой сегмент. Чтобы и в этом случае можно было бы построить оверлейную структуру, Компоновщик позволял определять области, в которые помещаются сегменты.

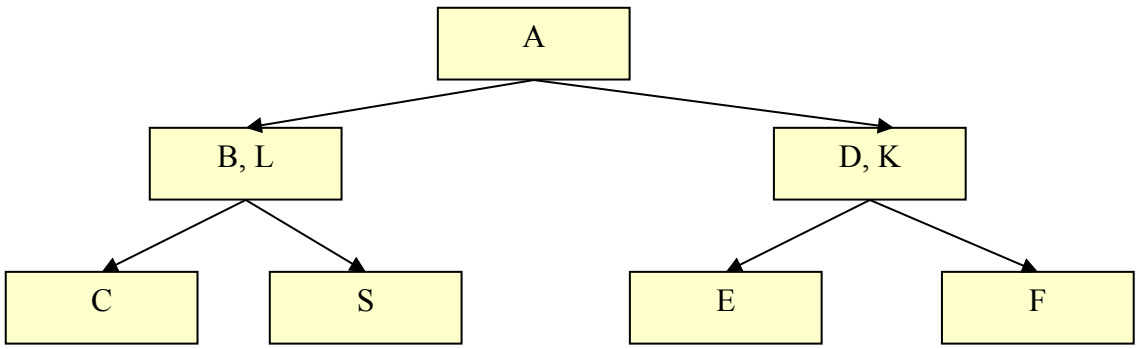


Рис. 38. Дерево сегментов оверлейного модуля.

Рассматривать все нюансы не имеет смысла, поскольку в современных системах, использующих виртуальную память, оверлейные структуры не используются.

5.3.3 Загрузочный модуль динамической структуры

В отличие от модулей оверлейной структуры модули динамической структуры широко используются в z/OS MVS. Также в Windows существуют динамически подключаемые библиотеки (DLL), которые можно компоновать как статически, перед выполнением модуля, но и динамически, то есть в процессе выполнения. Это дает возможность строить модули динамической структуры. В UNIX модулей динамической структуры также нет. Даже введение многопоточности в этих системах пошло по пути введение так называемых «легких процессов». То есть каждый поток (thread) рассматривается как процесс с ограниченными полномочиями, выполняющийся в рамках другого процесса.

Реализация поддержки модулей динамической структуры требует наличия в управляющей программе функций, которые и составляют систему управления программами. При программировании существуют макрокоманды, которые позволяют с помощью программного интерфейса обращаться к этим функциям. Наиболее развитая система управления программами существует в ОС IBM и сохранилась в современной системе для мэйнфреймов z/OS MVS. Следующие макрокоманды поддерживают программирование модулей динамической структуры: LOAD, DELETE, LINK, XCTL. Функции LOAD и DELETE используются в паре.

LOAD загружает копию модуля из библиотеки загрузочных модулей в область доступной памяти задачи и возвращает адрес точки входа. Если программный модуль загружается несколько раз, то реально загрузка осуществляется только первый раз и в блоке запроса увеличивается значение счетчика загруженных программ. В последующие выполнения функция LOAD проверяет значение этого счетчика в блоке запросов этой программы и, если значение больше нуля, то просто увеличивает его на 1. Во всех случаях возвращается адрес точки входа, но обращения к этой программе не происходит. Программист вызывает эту программу стандартным образом, используя команду процессора вызова подпрограмм, например, CALL, и прерывания не происходит. Во время выполнения макрокоманд используется программный интерфейс (см. Тему 4), основанный на синхронном программном прерывании, и осуществляется переход к функции ядра LOAD.

DELETE проверяет счетчик загруженных программ и, если его значение больше 0, то значение счетчика уменьшается на 1. Если после уменьшения, значение стало равным 0, то память в области задачи, занимаемая этой программой может быть освобождена.

LINK также загружает программный модуль в область памяти задачи, но сразу передает управление этому модулю. После завершения работы этот модуль может быть выгружен.

XCTL использовалась в начальных системах, в которых объем основной памяти был небольшой. Эта макрокоманда позволяла загружать программный модуль, передавать ему управление, но без возврата. С появлением систем виртуальной памяти необходимость в такой функции отпала.

Функции ядра для управления программами поддерживают в области системных очередей структуру данных, которая состоит из списка управляющих блоков, описывающих активные и загруженные в ОП программные модули. Эти блоки называются программными блоками запросов PRB (program request block). Они используются для регистрации активных программ Для учета загруженных программ используются блоки загруженных программ (LPRB – load program request block).

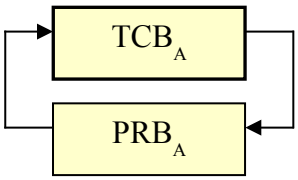


Рис. 40

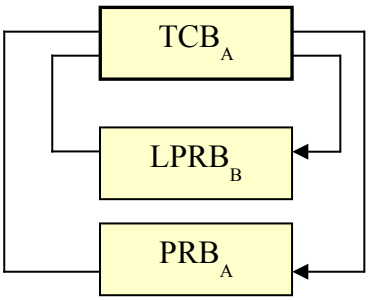


Рис. 41

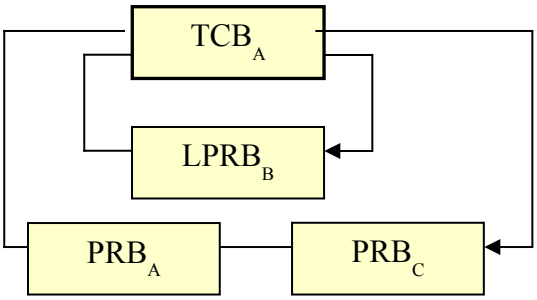


Рис. 42

```
A   START   0
    LOAD  B
    LINK   C
    CALL  B
    DELETE  B
    END
```

```
C   START   0
    LOAD  B
    CALL  B
    DELETE  B
    END
```

Рис. 39. Программный модуль
динамической структуры.

На рис. 39 приводится пример использования этих макрокоманд для построения модуля динамической структуры. Модуль А загружает модуль В, затем загружает и выполняет модуль С. Потом происходит выполнение загруженного модуля В и удаление его. Модуль С загружает модуль В, передает ему управление и затем выгружает его. На рис. 42 показано изменение структуры управляющих блоков, которые отражают состояние программ задачи А. Модуль А является головным модулем, с которого начинается выполнение. Имя этого модуля и берется в качестве имени всей задачи. Для выполнения программного модуля создается задача и строится в области системных очередей блок управления задачей (TCB – task control block). К этому блоку TCB присоединяется PRBA, который указывает на то, что загружена и исполняется программа А. Эта ситуация показана на рис. 40.

После выполнения макрокоманды LOAD и загрузки программы В строится LPRBb, отражающий загрузку модуля В. Счетчик загруженных программ в этом блоке устанавливается в единицу. Структура управляющих блоков после выполнения макрокоманды LOAD показана на рис.41.

При выполнении макрокоманды LINK загружается и исполняется программа С. Соответствующим образом и изменяется структура управляющих блоков. В список блоков активных программ добавляется блок PRB для программы С. Структура управляющих блоков показана на рис. 42.

Когда выполняется макрокоманда LOAD в программе С, то к счетчику загруженных программ добавляется 1, а загрузка не выполняется, поскольку модуль В уже находится в памяти. Адрес точки входа берется из блока LPRBB и предоставляется программе С. Этот адрес используется в команде CALL. Значение счетчика загруженных программ становится равным 2.

После выполнения программы В выполняется макрокоманда DELETE, которая вычитает из счетчика загруженных программ единицу. Поскольку оставшееся значение не 0, то блок LPRBB остается и структура управляющих блоков не меняется.

После завершения выполнения программы С управление передается в программу А и блок PRBc удаляется. В результате структура управляющих блоков преобразуется к показанной на рис. 41.

Затем опять вызывается программа В и после её окончания исполняется макрокоманда DELETE, которая вычитает из счетчика загруженных программ в блоке LPRBB единицу. В результате вычитания значение счетчика становится равным 0 и блок LPRBB удаляется. Структура управляющих блоков примет вид, показанный на рис. 42.

По завершении программы А задача будет завершена и все управляющие блоки, идентифицирующие программы и задачи в системе удаляются.

Таким образом, для управления программами ядро ОС содержит функции, позволяющие загружать в область задачи другие загрузочные модули и исполнять их. Доступ к этим функциям осуществляется с использованием программного интерфейса по программному прерыванию.

5.4 Организация программных модулей

В зависимости от применения программные модули могут иметь различную организацию. Для того чтобы получить тот или иной модуль необходимо соответствующим образом написать исходный код и установить соответствующие опции системы программирования. С точки зрения организации выделяют следующие модули:

1. Однократно используемый программный модуль.
2. Повторно используемый программный модуль.
3. Повторно входимый программный модуль.
4. Сопрограммы.

5.4.1 Однократно используемый программный модуль

В программных модулях этого типа некоторые переменные инициализируются не в процессе выполнения программы, а статически, в процессе написания. Некоторые языки поддерживают специальное описание для этих переменных. Так оператор DATA в языке ФОРТРАН позволяет задать начальные значения переменных. Например, строка

```
DATA N/107/
```

задает значение переменной N. Если в программе это значение нигде не восстанавливается, то измененное значение будет сохраняться. Пример такой программы на языке ФОРТРАН:

```
SUBROUTINE  RAND(L)
DATA  N/107/
N=25173*N+13849
N=N-(N/65536)*65536
L=N
RETURN
END
```

Каждый раз при обращении к подпрограмме RAND используется не начальное значение N, а текущее, оставшееся после предыдущего выполнения.

Другой пример на языке С использует описание static.

```
int rand(){
static int n = 107;
n=25173*n+13849;
n=n%65536;
```



```
return n;  }
```

Здесь также при каждом обращении к функции будет вычисляться новое значение поскольку будет браться значение n, измененное при предыдущем выполнении. Однако, если перезапустить программу, то есть если загрузить её с диска в память и выполнить, то будет вычислен ряд значений совпадающий с предыдущим запуском.

Рис. 43. Загрузочный модуль

Таким образом, копия загрузочного модуля на диске содержит поле переменной n (N) инициализированное начальным значением (107). При загрузке в основную память поле этой переменной также имеет начальное значение, но при выполнении программы это значение изменяется и не восстанавливается.

На рис. 43 приведен загрузочный модуль, который содержит инициализированную переменную N. Понятно, что при загрузке в ОП это инициализированное значение переносится, а затем изменяется.

Такой способ организации модуля может использоваться для программ генерирующих последовательность псевдослучайных чисел. При каждом обращении функция вычисляет очередное псевдослучайное число.

5.4.2 Повторно используемый программный модуль

Этот способ организации наиболее широко распространен в практике программирования. Он отличается от однократно используемых модулей тем, что инициализация переменных осуществляется во время выполнения. При каждом обращении к подпрограмме с одними и теми же значениями параметров результат будет тот же.

5.4.3 Повторно входимый программный модуль

Этот тип модуля появился в мультипрограммных системах. Другие названия этого типа модулей – реентерабельный, реентрантный, чистая процедура. Особенность такого модуля состоит в том, что его программный код выполняется (используется) двумя и более процессами. Он позволяет другому процессу прерваться при исполнении реентерабельного кода. Другой процесс начинает или продолжает выполнение этого кода как программы процесса. Затем он также может быть прерван. Управление получает первый процесс, который продолжает исполнять прерванный код реентерабельного модуля. Таким образом, можно загрузить только одну копию программного

кода, которую могут выполнять несколько процессов. На рис. 44 показан пример использования реентерабельной программы двумя процессами.

Для того, чтобы стал возможным такой способ исполнения программы, код должен быть написан определенным образом. Прежде всего, программный код должен не зависеть от того, в рамках какого процесса он исполняется. В коде не должно быть команд, которые модифицируют другие команды, не должно быть переменных, изменяющихся в процессе выполнения. Также в реентерабельном коде нельзя напрямую обращаться к данным типа static, нельзя вызывать нереентерабельные функции.

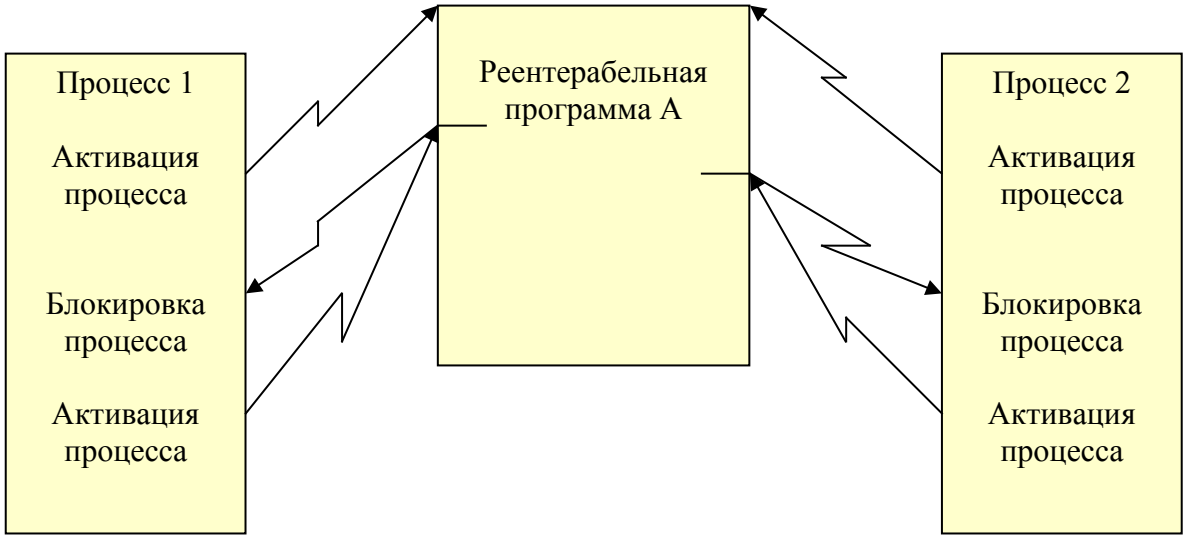


Рис. 44. Использование реентерабельной программы

Однако, в любой программе существуют элементы, зависящие от применения. Это стек, куча и переменные, изменяющиеся в процессе исполнения. Эта изменяющаяся часть программы у каждого процесса своя, индивидуальная. Поэтому возникает задача написания кода таким образом, чтобы при переключении процессов команды программы работали с той частью изменяющейся программы, которая принадлежит активному процессу. При переключении процессов перегружается контекст процесса, то есть значения регистров блокированного процесса запоминаются, а регистры загружаются значениями из контекста активного процесса. Если программный код содержит команды, в которых используются только относительные адреса, то перемена значений в регистрах, работающих со стеком, базовых регистров для данных и кучи позволяют обеспечить работу с данными нового процесса. Построение реентерабельного модуля зависит от архитектуры и адресации процессора.

Реентерабельные программы используются для написания функций ядра. Также в современных многопроцессорных системах при организации

многопотоковых приложений необходимо общие функции, используемые разными потоками, оформлять как реентерабельные программы. Здесь возникает такая же задача, как и в случае использования реентерабельной программы, несколькими процессами.

5.4.4 Сопрограммы

Техника сопрограмм появилась в связи с необходимостью в однопрограммной системе реализовать мультипрограммную обработку. То есть происходит моделирование логического параллелизма, когда выполняются участки различных программ в порядке, определяемом логикой самих программ.

Для реализации сопрограмм необходимо использовать систему программирования, позволяющую изготавливать многовходовые модули. Как правило, системы программирования на ассемблере и С обеспечивают такую возможность. Также необходимо изготовить функцию, которая осуществляет переход на заданную точку входа определенной программы и сохраняет адрес возврата. Пример сопрограммной структуры из двух программ показан на рис. 45.

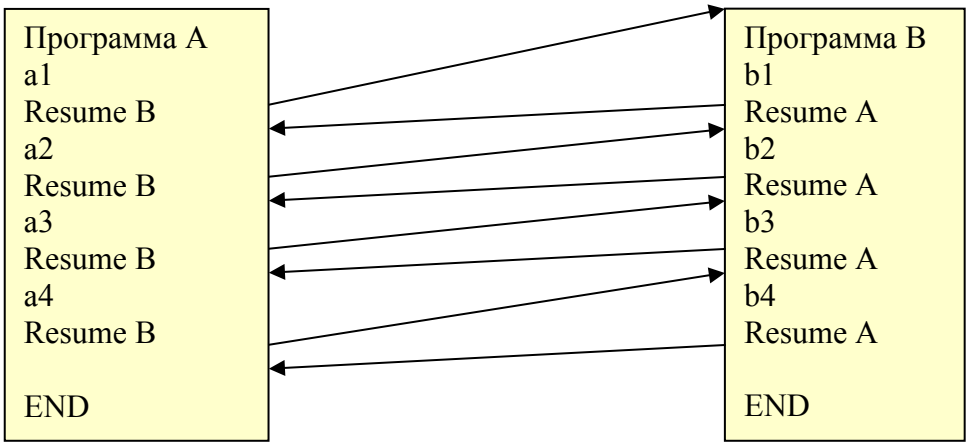


Рис. 45. Сопрограммная структура двух программ

Временная диаграмма выполнения сопрограммы показана на рис. 46. В разные моменты времени выполняются различные участки одной и другой программ. Логика выполнения участков фиксирована и определяется логикой выполнения программ.

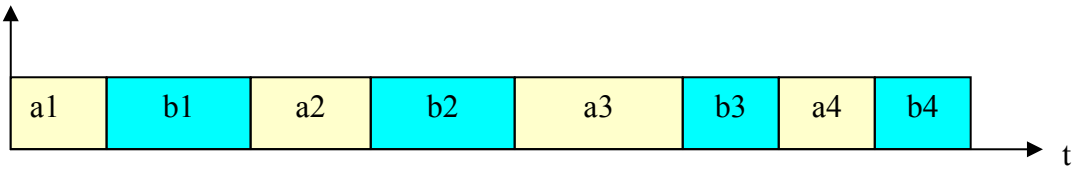


Рис. 46. Временная диаграмма выполнения сопрограммы

В современных мультипрограммных и мультипроцессорных системах такой способ организации программ, как правило, не используется. Управляющая программа обеспечивает функции взаимодействия и синхронизации процессов, используя которые программисты организуют обработку информации взаимодействующими процессами.

Часть II. ОРГАНИЗАЦИЯ И ФУНКЦИОНИРОВАНИЕ УПРАВЛЯЮЩЕЙ ПРОГРАММЫ

В этой части излагаемого материала операционная система понимается как система взаимодействующих процессов и ресурсов. Организация этого взаимодействия и управление процессами и ресурсами осуществляет управляющая программа. То есть в этой части операционная система понимается в узком специфическом смысле, что и является предметом изучения данного курса.

Управляющая программа представляет собой набор функций, которые резидентно размещаются в Основной Памяти (ОП) компьютера и образуют ядро. Это размещение происходит в момент начальной загрузки системы. Обращение к этим функциям происходит из программ процессов с использованием программного интерфейса.

Управляющая программа обеспечивает следующие основные функции:

1. Управление основной памятью.
2. Управление процессами.
3. Управление устройствами.
4. Управление вводом/вводом.
5. Управление данными.

Собственно, алгоритмы и реализация этих функций в различных типах операционных систем и являются предметом рассмотрения этой части.

Тема 6. ПОНЯТИЕ МУЛЬТИПРОГРАММИРОВАНИЯ

Под мультипрограммированием или мультипрограммным режимом работы компьютерной системы (КС) понимается такой режим, при котором одновременно в системе выполняется несколько независимых работ по переработке информации. Эти работы оформлены в виде процессов или задач. Название зависит от принятой в системе терминологии. Так в юниксоподобных системах (называемых Open system) это будет именоваться процессом, а в z/OS используется термин «задача».

С мультипрограммным режимом работы связаны такие понятия как уровень мультипрограммирования и смесь задач.

Уровень мультипрограммирования – число задач, выполняющихся одновременно в КС.

Смесь задач – качественная характеристика состава задач, выполняющихся одновременно, которая влияет на загрузку ЦП. Эта характеристика связана со свойствами выполняемых задач. Это свойство определяется процентом времени выполнения задачи, отводимого на операции с внешними устройствами, то есть все время выполнения задачи можно разделить на ту часть, когда задача занимает ЦП, и часть, когда она занимается вводом/выводом и освобождает ЦП. Понятие смеси задач поясним на примере двух задач, которые выполняются в мультипрограммном режиме. Пусть одна из этих задач занимается расчетами, то есть мало времени тратит на ввод/вывод, а много времени занимает ЦП. Назовем эту задачу счетной. Другая задача наоборот, занимает мало времени ЦП, но активно работает с внешними устройствами. Назовем эту задачу информационной. Выполнение этих двух задач позволит загрузить и ЦП, и внешние устройства, поскольку счетная задача будет иметь возможность занимать ЦП в те интервалы времени, когда информационная задача осуществляет работу с внешними устройствами. Главное обеспечить информационной задаче возможность занимать центральный процессор своевременно для запуска очередных операций с внешними устройствами.

Если бы в смеси присутствовали только счетные задачи, то внешние устройства бы простаивали, а процессор был бы загружен. Если бы в смеси присутствовали только информационные задачи, то ЦП бы простаивал. Таким образом, формирование оптимальной смеси задач улучшает производительность ВС.

6.1 Аппаратные средства поддержки мультипрограммирования

Реализация мультипрограммирования предполагает, что в основной памяти размещаются адресные пространства нескольких процессов. В этом случае компьютер должен обладать средствами защиты памяти, поскольку необходимо предотвращать несанкционированный доступ в адресные пространства других процессов и в область памяти управляющей программы.

В однозадачных системах защита памяти, как правило, не использовалась. Если в результате ошибок в приложении портилась область памяти управляющей программы, то просто перегружали ОС. Страдало само приложение, но оно и так работало некорректно. На других пользователей это не оказывало никакого влияния, их просто нет.

Иное дело при мультипрограммировании. Перезагрузка системы является чрезвычайным обстоятельством и приводит к прекращению выполнения корректно работающих приложений других пользователей. Поэтому, если какое-то приложение работает неправильно, то это не должно влиять на других пользователей и на работоспособность системы.

Таким образом, приложения работают в своих адресных пространствах и их доступ к адресам памяти ограничен. Функции управляющей программы могут обращаться к любым адресам памяти.

Кроме защиты памяти в мультипрограммных системах необходимо распределять ресурсы между процессами. Поручить это процессам невозможно по ряду обстоятельств. Во-первых, могут возникнуть коллизии между процессами, когда два или более процессов захватят один ресурс, который не может принадлежать двум процессам одновременно. Во-вторых, поручить прикладным программистам, писать код для работы с ресурсами просто невозможно. Поэтому, в мультипрограммной системе необходимо обеспечить централизованное распределение ресурсов.

Для того, чтобы управляющая программа могла осуществлять управление распределением необходимо запретить приложениям непосредственно обращаться к ресурсам. А функции управляющей программы могли бы это делать.

Для этого в процессоре выделяют два режима работы. Режим супервизора или режим управляющей программы позволяет использовать выполняющейся программе полный набор инструкций процессора. В этом режиме выполняются функции ядра и системные процессы. Другой режим позволяет использовать ограниченный набор инструкций и называется прикладным. Если встретится недопустимая инструкция, то возникает прерывание и при-

ложение заканчивается по ошибке. Этот режим используется для прикладных программ.

Для того, чтобы получить доступ к ресурсам, прикладные программы используют программный интерфейс, который основан на программном (синхронном) прерывании. При выполнении этого прерывания управление передается управляющей программе и осуществляется смена режима процессора на режим супервизора. При возврате в прикладную программу режим процессора также изменяется на прикладной.

Таким образом от аппаратных средств требуется поддержка реализации мультипрограммной ОС. Современные процессоры не только обеспечивают общую поддержку в виде той, что описана выше, но и специфическую поддержку конкретных широко используемых ОС, таких как UNIX или Linux.

6.2 Оценка загрузки центрального процессора в зависимости от уровня мультипрограммирования

Важный вопрос при применении мультипрограммирования состоит в оценке времени загрузки процессора. Естественно полагать, что чем больше загружен процессор, тем более производительно работает система. Правда, это справедливо для оценки производительности, измеряемой в количестве выполненных задач в единицу времени, то есть для пакетных ОС. Однако, эти рассмотрения позволяют оценить и загрузку процессора в случае диалоговых систем.

Для получения необходимых соотношений строится модель размножения и гибели, которая описывает состояния системы.

Пусть в системе находится n процессов. Система находится в состоянии Si, если i процессов находятся в состоянии ожидания завершения ввода-вывода, то есть блокированы. Тогда

S0 – состояние системы, когда все процессы разблокированы, то есть активны.

Sn – состояние системы, когда все процессы блокированы и процессор простаивает.

Диаграмму состояний системы будет выглядеть следующим образом (рис.47).

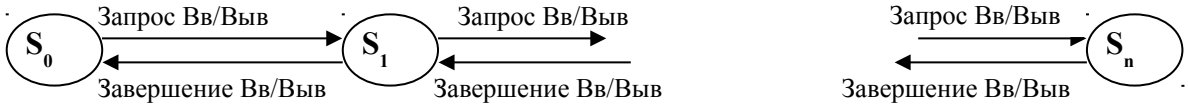


Рис. 47. Диаграмма состояний системы

Система переходит из состояния Si в состояние Si+1, если какой-то процесс запросил ввод/вывод и перешел в состояние ожидания завершения ввода/вывода, то есть блокировался. Также система переходит из состояния Si+1 в состояние Si, если для какого-то процесса ввод/вывод завершился, и процесс был разблокирован.

Пусть Pi вероятность пребывания системы в состоянии Si. Тогда Pn вероятность пребывания системы в состоянии Sn. Таким образом, это та вероятность, которая будет характеризовать простой процессора, поскольку все процессы блокированы.

Для построения вероятностного процесса работы системы положим, что λdt – вероятность блокирования процесса в результате запроса на ввод/вывод к моменту следующего наблюдения. А μdt – вероятность завершения ввода/вывода к следующему моменту времени наблюдения. Тогда вероятностная модель системы будет выглядеть следующим образом (Рис. 48).

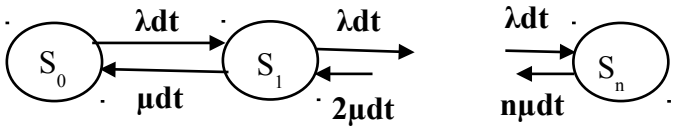


Рис.48. Вероятностная модель системы

Переход из состояния Si+1 в состояние Si равен (i+1)μdt, поскольку завершение ввода/вывода являются независимыми событиями и вероятность перехода равна сумме вероятностей завершения (i+1) операций ввода/вывода.

Для того чтобы характеризовать процесс, вводится величина ω, которая отражает вероятность ожидания ввода/вывода при выполнении процесса без мультипрограммирования. За время своего выполнения процесс переходит из активного состояния в состояние ожидания и, затем, опять возвращается в активное. В активном состоянии программы выполняются на процессоре, а в блокированном нет. Времена нахождения процесса в активном состоянии и блокированном удобнее рассматривать как случайные величины. Даже для одного и того же процесса эти времена меняются в зависимости от исходных данных и условий выполнения. Тогда доля времени, в течении которого процесс находится в состоянии блокирования, ожидая окончание ввода/вывода будет:

$$\omega = \lambda dt/(\lambda dt + \mu dt) = \lambda/(\lambda + \mu) \quad (1)$$

Если значение λ велико, то это соответствует частым запросам на ввод/вывод. В этом случае система неизбежно перейдет в состояние Sn с вероятностью 1. Если μ велико, то это соответствует ситуации, когда устройства ввода/вывода работают быстро и система перейдет в состояние S_0 с вероятностью 1. Поэтому следует рассмотреть ситуацию, когда наблюдается

некоторое равновесие, то есть число переходов из состояния Si в состояние Si+1 равно числу переходов из состояния Si+1 в состояние Si. Это условие называется условием стационарности. В вероятностной модели условие будет выглядеть как равенство вероятностей перехода системы из состояния Si в состояние Si+1 и переходов из состояния Si+1 в состояние Si. Вероятность перехода из состояния i в состояние i+1 равна произведению вероятности перехода из Si в Si+1 и вероятности нахождения системы в состоянии Si. Это уравнение выглядит следующим образом:

$$\mathbf{p_{i,i+1}P_i = p_{i+1,i}P_{i+1}} \quad (2)$$

Используя это выражение, подставим в это уравнение вероятности из модели и получим систему уравнений:

$$\begin{gathered}\mathbf{\lambda dt\ P_0 = \mu dt\ P_1}\\ \mathbf{\lambda dt\ P_1 = 2\mu dt\ P_2}\\ \mathbf{\ldots\ldots\ldots\ldots\ldots}\\ \mathbf{\lambda dt\ P_{n-1} = n\ \mu dt\ P_n}\end{gathered} \quad (3)$$

Из этой системы уравнений получим выражение для P_i через P_0.

$$\mathbf{P_1 = (\lambda/\mu)\ P_0}$$

$$\mathbf{P_2 = (\lambda/2\mu)\ P_1 = ((\lambda/\mu)^2/2)\ P_0}$$

$$\mathbf{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots}$$

$$\mathbf{P_i = ((\lambda/\mu)^i/i!)\ P_0} \quad (4)$$

$$\mathbf{\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots}$$

$$\mathbf{P_n = ((\lambda/\mu)^n/n!)\ P_0} \quad (5)$$

Заметим, что $\sum_{i=0}^{n} Pi = 1$

Откуда $P_0+\sum_{i=1}^{n} Pi = 1 \quad P_0=1-\sum_{i=1}^{n} Pi$

Подставим под знак суммы выражение для Pi из (4).

$$\mathbf{P_0=1-\sum_{i=1}^{n}\left((\lambda/\mu)^i/i!\right)\ P_0}$$

Это уравнение решим относительно P0

$$\mathbf{P_0=1/\left(1+\sum_{i=1}^{n}\left((\lambda/\mu)^i/i!\right)\right)}$$

Для того, чтобы привести полученное выражение к более удобному виду, используем равенство:

$$\mathbf{\left((\lambda/\mu)^0(1/0!) = 1\right.}$$

и получим

$$P_0=1/(\sum_{i=0}^{n}((\lambda/\mu)^i/i!))$$

В выражение (5) для Pn подставим полученное выражение для P0 и получим:

$$P_n=((\lambda/\mu)^n)/n!(\sum_{i=0}^{n}((\lambda/\mu)^i/i!)) \quad (6)$$

Теперь необходимо выразить это соотношение через ω. Выражение (1) преобразуем к виду:

$$\lambda/\mu = \omega/(1 - \omega)$$

Тогда выражение (6) примет окончательный вид (7).

$$P_n=((\ \omega/(1 - \omega))^n)/n!(\sum_{i=0}^{n}((\omega/(1 - \omega))^i/i!)) \quad (7)$$

Напомним, что Pn это вероятность нахождение системы в состоянии, когда все процессы блокированы и ожидают окончания ввода/вывода. В этом состоянии процессор простаивает. В былые времена это было настолько важно, что в те моменты времени, когда процессор был на холостом ходу, на пульте оператора загоралась красная лампочка.

Используя выражение (7) можно оценить процент простоя процессора, выразив значение Pn в процентах, от уровня мультипрограммирования n и процента времени ввода/вывода задачи ω.

Данные расчета приведены в таблице 5.

Если задачи требуют только 25% времени для выполнения ввода/вывода, то они занимают процессор достаточно много времени и при увеличении числа задач процессор будет все время загружен, и время простоя стремится к нулю. Напротив, если задачи требуют много времени для ввода/вывода, то даже высокий уровень мультипрограммирования не позволяет полностью загрузить процессор.

Таблица 5

n \ ω %	25	50	75
2	4.0	20.0	52.9
3	0.4	6.3	34.6
4	0	1.5	20.6
5	0	0.3	11.0

Построенная модель достаточно хорошо описывает поведение системы и позволяет оценить загрузку системы.

Тема 7. УПРАВЛЕНИЕ ОСНОВНОЙ ПАМЯТЬЮ В ОС

Организация управления ОП в КС тесно связана с архитектурой аппаратных средств и поддерживается специальными механизмами, реализованными в аппаратуре. На организацию системы управления ОП влияют:

1. организация памяти в компьютере,
2. механизмы защиты памяти,
3. механизмы адресации.

Функции управления ОП располагаются резидентно в ядре ОС. Их используют как пользовательские и системные программы, так и другие модули ядра. Реализация управления ОП для каждой КС весьма специфична и зависит от возможностей аппаратуры и стратегии распределения памяти, то есть принципа, в соответствии с которым выделяется память под различные задачи.

Система управления ОП выполняет следующие функции:

1. учет ОП в ОС,
2. планирование запросов на выделение свободных участков ОП,
3. выделение и освобождение ОП.

Учет состояния памяти заключается в отслеживании занятых и свободных областей памяти в каждый момент времени посредством изменения информации в системных структурах данных.

Планирование запросов на предоставление памяти состоит в определении задачи или программы, получающей память, момента времени, в который выделяется память, в определении объема выделяемой памяти.

Выделение и освобождение памяти связаны с работой алгоритмов поиска свободных участков памяти требуемого объема, коррекцией системных структур данных и предоставлением информации о выделенном участке памяти в ответ на запрос.

Реализация этих функций существенно зависит от аппаратной части, а также от назначения ОС. Рассмотрению стратегий управления памятью, способов реализации функций посвящена эта тема.

7.1 Связывание загрузочного модуля с адресами ОП

Логическое адресное пространство - совокупность программных адресов, которые занимают программы и данные задачи.

Физическое адресное пространство задачи - совокупность абсолютных адресов физической памяти, в которых располагается задача.

Система управления ОП осуществляет отображение логического адресного пространства задачи в физическое. Этот процесс состоит из поиска и выделения свободных участков физической памяти, загрузки копии программы с диска в ОП и связывания программных адресов с физическими. Загрузку копии осуществляет модуль ядра - загрузчик. Он же настраивает программные адреса на физическую память в случае статического связывания. При использовании динамического связывания преобразование программного адреса в абсолютный адрес осуществляется аппаратными средствами.

Статическое связывание – назначение корректных физических адресов в командах загруженной программы, использующих прямую адресацию. Лучше всего это рассмотреть на весьма популярном примере. Когда на ассемблере для MS DOS создается загрузочный модуль типа .exe , то необходимо в первых командах кода написать

```
mov   AX, seg DATA
mov   DS, AX
```

Второй операнд первой команды задает прямой адрес – сегментный адрес сегмента DATA. Этот адрес в момент компиляции неизвестен и в файле копии загрузочного модуля на диске корректное значение адреса отсутствует. В начале файла копии загрузочного модуля на диске помещается таблица, которая содержит указания на поля операндов в командах, которые требуют настройки при перемещении копии загрузочного модуля в ОП, а также описание этих адресов. Используя эту информацию, загрузчик корректирует ад-

реса, поскольку их значения ему известны. Все эти действия загрузчик выполняет до передачи управления загрузочному модулю, поэтому подобный способ связывания называется статическим.

Динамическое связывание – вычисление действительных адресов в процессе выполнения загрузочного модуля аппаратными средствами процессора. Такой способ применяется в случае относительной, косвенной, страничной, сегментной и сегментно-страничной способах адресации.

Логическое адресное пространство может быть односегментным и многосегментным. Под логическим сегментом понимается логически законченный объем информации (программа, подпрограмма, структура данных, объект класса и т.д.). Широко используется понятие физического сегмента, под которым понимают некоторый объем физической памяти.

Физическое адресное пространство может быть одноуровневым и многоуровневым. Многоуровневое физическое адресное пространство представляет собой иерархию уровней памяти. Как правило, выделяют сверхоперативную, оперативную, массовую и внешнюю памяти. Уровни памяти различаются по своим временным, емкостным и стоимостным характеристикам. Размещение задач и адресного пространства одной задачи, динамическое управление этим размещением позволяют существенно повысить пропускную способность КС.

Компьютерные системы, имеющие одноуровневую организацию физического адресного пространства ОП, называются нестраничными. При многоуровневой организации физического адресного пространства используются системы виртуальной памяти. Иногда такой способ называют страничной памятью. В настоящем изложении под страничной памятью будет пониматься вполне определенный способ организации физической памяти.

7.2 Управление ОП в нестраничных системах

Общей особенностью стратегий управления ОП в нестраничных системах является то, что все логическое адресное пространство задачи загружается в физическую память. В результате чего, мало используемые функции, такие, например, как функции обработки ошибок, загружаются в ОП и память используется нерационально. Однако, эти методы управления памятью требуют меньших затрат на реализацию системы адресации процессора, более простые в реализации. К методам управления нестраничной памятью относятся:

одиночное непрерывное распределение,

распределение статическими разделами,

распределение динамическими разделами.

7.2.1 Одиночное непрерывное распределение

Стратегия одиночного непрерывного распределения используется в однопрограммных ОС. Она состоит в следующем.

После загрузки ОС вся оставшаяся ОП используется для задач, выполняющихся по командам пользователя. В каждый момент времени выполняются программы только одной задачи. Поэтому вся свободная память предоставляется этой задаче.

В таких системах управление ОП на уровне задач практически отсутствует, но предоставляются возможности управления памятью на уровне программ. Для этого в ядре содержатся функции выделения и освобождения памяти, которые используются в программах при их разработке. Таким образом, функции управления памятью осуществляет программист. На рис. 49 представлен пример одиночного непрерывного распределения памяти. Область памяти управляющей программы находится в младших адресах. Остальная память отводится задаче.

Рис. 49 Одиночное непрерывное распределение памяти

Основные функции управления памятью выполняются на уровне программ, а не задач. Так функция учета заключается в построении списка блоков управления памятью MCB (memory control block). Адрес первого блока этого списка находится в области управляющей программы. В начале каждого участка памяти находится MCB, в котором содержится описание этого участка: размер, занят или свободен, какой программой или данными занят участок.

Управление этим списком осуществляется функциями выделения и освобождения памяти. Планирование запросов на выделение и освобождение памяти реализует программист в момент разработки программы.

Конечно, даже в однозадачной ОС необходимо защищать область памяти управляющей программы от вмешательства программ пользователя. Это можно делать с помощью регистра защиты, в котором хранится адрес начала области памяти, предоставляемый задаче. При вычислении абсолютного адреса аппаратные средства должны сравнивать его значение с содержимым регистра защиты. Если значение абсолютного ад-

реса лежит в области управляющей программы, то возникает прерывание по защите памяти. Однако в реальных ОС, использующих такую стратегию управления памятью, как правило, защита области УП не осуществляется. Это объясняется тем, что порча области памяти УП происходит в момент выполнения задачи пользователя. Особенно часто это случается при отладке программ. Поскольку в однопрограммных ОС программы других пользователей в этот момент времени не содержатся в ОП, то страдает от несанкционированных действий сам нарушитель. Восстановление системы, как правило, осуществляется ее перезагрузкой.

Иное дело в мультипрограммных системах. Несанкционированный доступ к области УП или областям памяти задач других пользователей может отразиться на их работе, поэтому такое вмешательство недопустимо и области памяти должны быть защищены.

Достоинством стратегии одиночного непрерывного распределения ОП является простота реализации.

Недостатки состоят в следующем:

1) невозможно реализовать мультипрограммирование. При достаточно мощном процессоре и больших объемах ОП применять эту стратегию нецелесообразно, так как память будет использоваться плохо, а процессор будет простаивать. Данный способ пригоден для маломощных небольших КС;

2) этой стратегии присуще плохое использование памяти. Часть памяти не используется совсем, так как объем загружаемой задачи обычно меньше объема свободной ОП. Кроме того, поскольку в физическую память загружается все логическое адресное пространство задачи, то загружаются и редко исполняемые участки кода. К ним можно отнести, например функции обработки ошибок;

3) отсутствие защиты областей ОП программных модулей, присущее стратегии одиночного непрерывного распределения приводит к существенным трудностям при реализации сложных программных проектов. В процессе отладки одни модули портят память других, и такие ошибки весьма трудно отыскиваются.

В качестве примера управления ОП, организованной по такой стратегии, рекомендуется разобрать систему управления ОП MS DOS.

7.2.2 Распределение разделами

Стратегия распределения памяти разделами позволяет реализовать мультипрограммный режим работы. Задаче выделяется участок памяти – раздел.

Поскольку принцип распределения памяти разделами используется в мультипрограммных системах, и задачи различных пользователей выполняются в разных разделах, то должен быть исключен доступ задачи к другим разделам, то есть аппаратные средства должны обеспечивать защиту разделов и области памяти управляющей программы. Существуют способы защиты регистрами и ключами.

Защита регистрами заключается в реализации в процессоре пар регистров: базового регистра и регистра защиты. В базовый регистр загружался базовый адрес раздела памяти, отведенного задаче, а в регистр защиты загружался размер раздела. Таким образом, определялось адресное пространство раздела. При вычислении исполнительного адреса в программе, размещенной в этом разделе, этот адрес проверялся на принадлежность адресному пространству раздела. Если адрес не принадлежал разделу, то возникало прерывание по защите памяти. Так в процессоре DEC PDP11 было 16 пар таких регистров. Восемь из них могли использоваться управляющей программой, а восемь прикладной задачей. Таким образом, каждая задача могла размещаться в восьми разделах, каждый из которых был защищен своей парой регистров.

Защита ключами основана на том, что ОП выделяется определенными порциями. Так в компьютерах IBM память выделялась блоками по 4Кб. С каждым блоком был связан четырех байтный регистр защиты. При создании задачи ей присваивался код защиты, который записывался в Слово Состояния Программы (PSW) и регистры защиты тех блоков памяти, которые принадлежали этой задаче. При вычислении исполнительного адреса, код защиты в PSW сравнивался с кодом защиты в регистре защиты блока памяти, которому принадлежал исполнительный адрес. Если коды не совпадали, то возникало прерывание по защите памяти. Поскольку на хранение кода защиты отводилось 4 разряда, то коды защиты задач могли принимать значения от 1 до 15. Нулевой код приписывался системным задачам, и этот код обеспечивал доступ к любому блоку памяти.

Распределение статическими разделами. Принцип распределения ОП статическими разделами состоит в том, что вся свободная память после загрузки ОС делится на разделы фиксированной длины. В эти разделы и загру-

жаются задачи для выполнения. Это наиболее простой способ управления памятью, позволяющий организовать мультипрограммный режим работы. Данная стратегия управления ОП использовалась в первых мультипрограммных системах. Также она популярна в управляющих ВС, поскольку характеристики задач, порядок их выполнения, способы загрузки в память частично определяются в процессе проектирования.

Рассмотрим в качестве примера пакетную мультипрограммную ОС IBM OS360 MFT, в которой была использована подобная стратегия управления памятью. ОП делилась на разделы фиксированного размера, которые для данной версии, установленной на конкретном компьютере, были постоянны (рис. 50). Каждый раздел соответствовал классу входных работ и все задачи всех заданий, принадлежащих этому классу, выполнялись в этом разделе ОП.

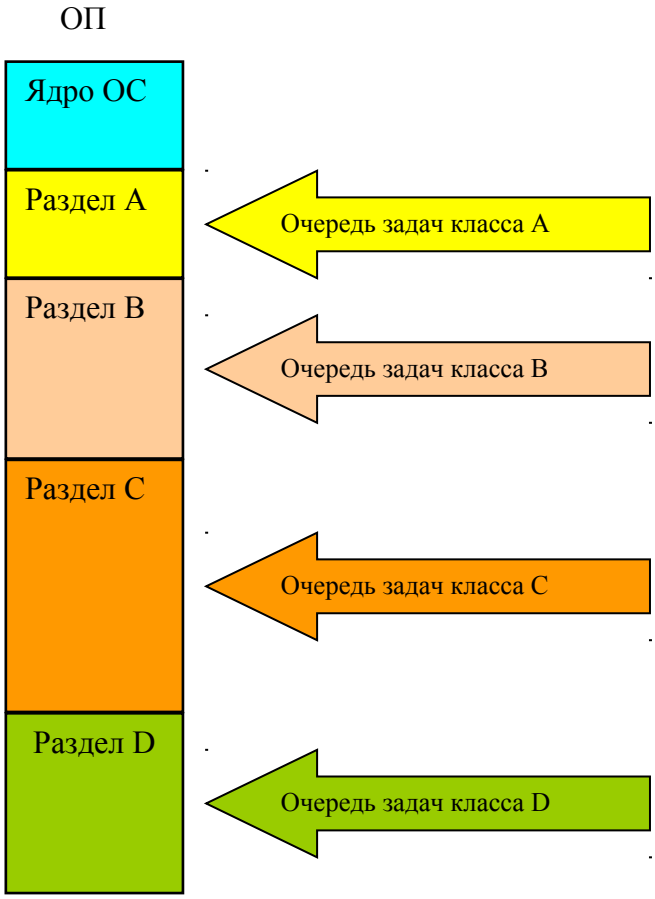


Рис. 50. Распределение ОП статическими разделами

Каким же образом задания попадали в соответствующий класс?

В операторе JOB программист указывал параметр CLASS=A и в, соответствии со значением этого параметра, задача Системного ввода помещала это задание в очередь класса A. Программист мог указать и любой другой класс, в зависимости от характеристик своей задачи и информации о параметрах классов, которые существовали на используемом компьютере. Администрация вычислительного центра предоставляла такую информацию, характеризуя каждый класс размером раздела основной памяти и временем счета.

Такая организация системы позволяла сформировать хорошую смесь задач, поскольку задачи одного класса выполнялись последовательно, а задачи разных классов и создавали мультипрограммную смесь задач.

Стратегия распределения ОП статическими разделами также использовалась в ОС реального времени фирмы Digital Equipment Corporation (DEC), таких как RSX11 для компьютеров PDP11. В этом случае разделы имели имена и задачи загружались в соответствующие разделы. При создании задачи программист указывал, в какой раздел и как должна загружаться разрабатываемая задача. Эта информация хранилась в заголовке задачи и использовалась управляющей программой.

В настоящее время такой подход к управлению ОП используется во встроенных системах типа телефонов и другой аппаратуры.

Реализация функций управления ОП в таких системах достаточно проста.

Функция учета свободных и занятых разделов может быть реализована с помощью битового вектора. Каждый бит соответствует разделу и показывает, свободен раздел или занят.

Функция планирования выделения памяти для задачи связано с определением возможности загрузки задачи в раздел. В системах общего пользования к каждому разделу формируется очередь задач. Все задачи выполняются в порядке очереди. Момент времени выполнения определяется готовностью задачи, то есть задаче должны быть предоставлены необходимые для выполнения ресурсы.

Объем памяти, предоставляемый задаче, соответствует размеру раздела. Освобождается также вся память раздела.

Блокам памяти раздела, который отводился задаче, присваивался ключ задачи. Этот же ключ присваивался и другим ресурсам, выделенным задаче.

Основные достоинства стратегии распределения статическими разделами состоят в возможности реализации мультипрограммного режима работы и простоте реализации.

К недостаткам можно отнести неэффективное использование памяти, обусловленное различными размерами задач и неизвестной частотой их поступления. Если задачи одного класса поступают редко, то раздел, предназначенный для задач этого класса, будет использоваться плохо. Кроме того, для решения задач большого объема требуется соответствующий раздел. Но большие задачи встречаются более редко, чем маленькие задачи. Следовательно, раздел большого объема будет пустовать. Чтобы компенсировать этот недостаток, разрешается одной очереди задач приписывать два раздела. Однако память в разделе большого объема будет использоваться неэффективно при загрузке в него маленькой задачи.

Способ распределения статическими разделами можно использовать, если заранее известны размеры задач и частота их поступления в компьютерную систему.

Поскольку в раздел загружается все адресное пространство задачи, то редко используемые программы будут занимать память. Это также не улучшает использование памяти.

Распределение динамическими разделами. Стратегия управления ОП, распределяемой динамическими разделами, является весьма распространенной в нестраничных системах. При распределении динамическими разделами память выделяется по запросам. В запросе содержится объем требуемого участка памяти. Запросы выдаются программой управления задачами при создании очередной задачи. Система управления памятью при удовлетворении запроса находит свободный участок требуемого объема и создает раздел задачи. При завершении задачи этот раздел уничтожается.

Такой способ управления ОП использовался в ОС IBM OS360 MVT, в версиях UNIX для DEC PDP11. Также в DEC RSX11M можно было создать раздел, в котором использовался этот метод, что позволяло использовать эту ОС как диалоговую многопользовательскую систему.

На рис. 51 показано изменение состояния памяти в процессе поступления запросов.

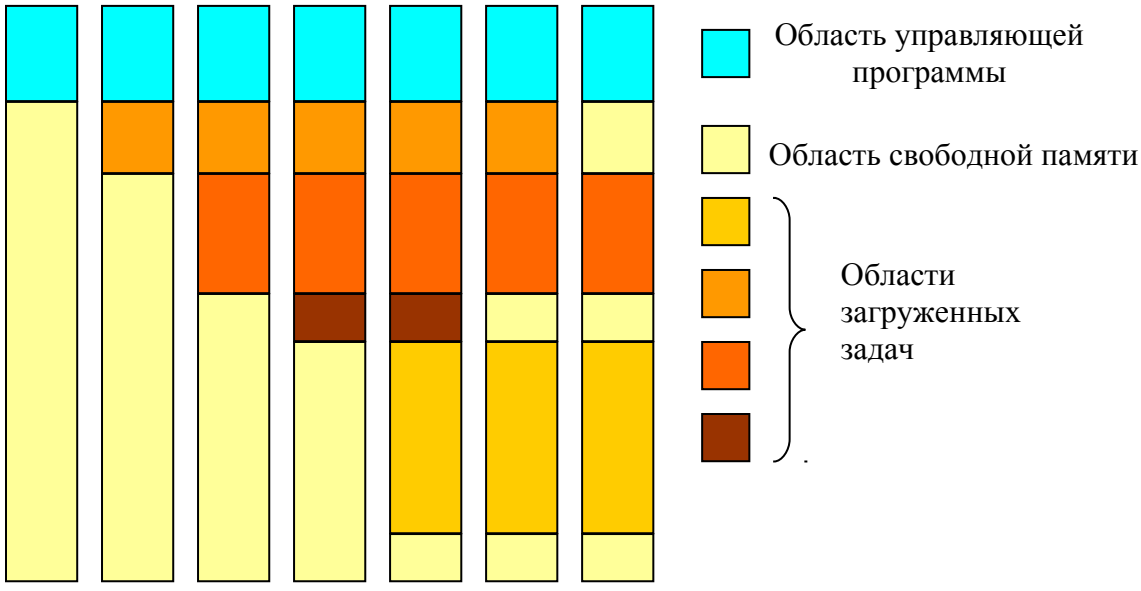


Рис. 51 Изменение состояния памяти при загрузке и удалении задач

При способе управления памятью, распределяемой динамическими разделами, необходимо учитывать свободные и занятые участки, то есть хранить информацию о базовых адресах и размерах этих участков памяти. Во

многих ОС для этой цели используется список блоков управления памятью. В области управляющей программы по определенному адресу размещается указатель начала списка, который содержит ссылку на первый блок.

В начало каждого свободного участка помещается блок управления памятью (MCB). В простейшем случае этот блок должен содержать адрес следующего свободного участка и объем участка, которым он управляет. MCB последнего свободного участка содержит 0 в поле адреса следующего участка.

С этим списком работают функции ядра, занимающиеся выделением и освобождением памяти. Для освобождения памяти необходимо знать адрес и размер занятого участка. Если в ОС принят способ учета только свободных участков, то информация о занятых участках хранится в блоке управления задачей. При завершении задачи освобождается занятый участок, и эта информация используется для реорганизации списка блоков свободной памяти. Можно с помощью списка учитывать как свободные, так и занятые участки памяти. Однако поиск свободного участка по такому списку занимает больше времени. На рис. 52 представлен список свободных блоков.

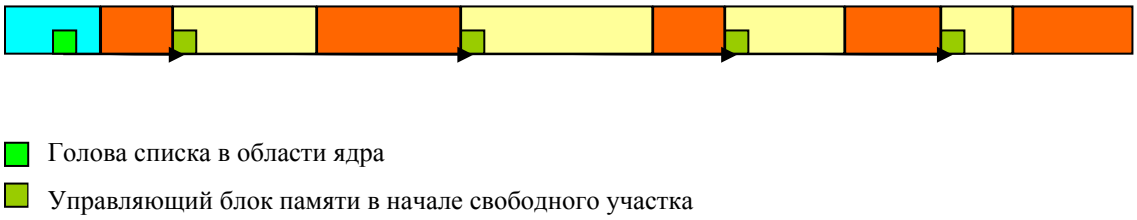


Рис. 52 Список свободных блоков

Планирование запросов на выделение памяти осуществляется функциями планирования задач в соответствии с реализованным в ОС алгоритмом. Так в первых версиях ОС UNIX право на загрузку в ОП получала задача дольше всех находившаяся в области свопинга. Функции планирования задач обращаются к загрузчику, который и выдает запрос на выделение участка памяти для очередной созданной задачи или загружаемой из области свопинга. Выполнение функций выделения и освобождения памяти связано с работой со списком.

Алгоритмы выделения памяти. Для того, чтобы выделить участок памяти по запросу необходимо найти свободный участок, размер которого больше или равен размеру запроса. На рис. 53 показаны два способа выделения участка.

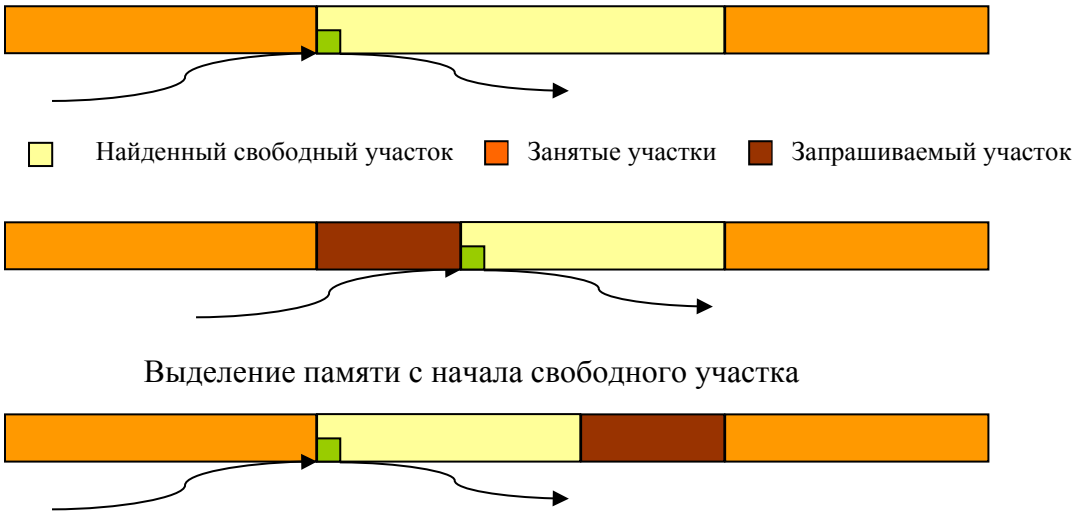


Рис. 53 Способы выделения участка

Можно выделять память с начала свободного участка. В этом случае придется изменять адрес предыдущей ссылки на новый свободный участок и формировать новый управляющий блок. Если отводить память под запрашиваемый раздел в конце свободного участка, то придется лишь изменить размер измененного свободного участка.

Существует три наиболее популярных алгоритма поиска свободного участка.

1. Алгоритм «Первый подходящий» (first fit)

Исходными данными для алгоритма служит размер запрашиваемого участка памяти Vp. Алгоритм просматривает список управляющих блоков свободных участков памяти и выбирает первый участок, чей размер $Vi \geq Vp$.

Равенство размеров запрашиваемого и выделяемого участков памяти реализуется редко, поэтому остается свободный участок. В начале списка группируются мелкие свободные участки, а свободные участки больших размеров сдвигаются в конец списка.

С одной стороны это положительный фактор, поскольку запросы на малые участки удовлетворяются в одной области памяти, а на большие – в другой. Это уменьшает дробление памяти на мелкие участки – фрагментацию. С другой стороны увеличивается время поиска свободного участка.

2. Алгоритм «Самый подходящий» (the best fit)

Алгоритм просматривает список управляющих блоков свободных участков памяти и выбирает участок, чей размер $Vi \geq Vp$ и разность Vi – Vp

минимальна, то есть выбирается наиболее близкий по размеру свободный участок, чей размер превосходит размер запроса.

Список свободных участков может быть упорядочен по возрастанию, тогда алгоритм «Самый подходящий » будет работать как «Первый подходящий» с этим списком. Но в этом случае при образовании нового свободного участка требуется сортировать список по возрастанию размеров разделов.

Достоинства этого алгоритма состоят в следующем:

1) если есть свободный участок, чей размер в точности равен размеру запроса, то этот участок будет выделен,

2) свободные участки больших размеров остаются нетронутыми.

Недостатки метода:

1) увеличивается вероятность образования свободных участков маленьких размеров, то есть увеличивается тенденция к фрагментации ОП,

2) при образовании новых свободных участков необходимо сортировать список, меняя ссылки.

3. Алгоритм «Наименее подходящий» (the worst fit)

Алгоритм просматривает список управляющих блоков свободных участков памяти и выбирает участок, чей размер $Vi \geq Vp$ и разность $Vi - Vp$ максимальна, то есть выбирается максимальный по размеру свободный участок.

Если список организован таким образом, что первым в списке находится максимальный свободный участок, то удовлетворение запроса осуществляется чрезвычайно просто – выбирается первый участок в списке. Однако при образовании нового свободного участка необходимо в списке отыскивать максимальный свободный участок.

Достоинство этого алгоритма состоит в том, что остаток после выделения участка по запросу остается большим.

Освобождение памяти.

Освобождение занятого участка памяти происходит при удалении задачи из ОП в результате ее окончания или перемещения ее в область свопинга на диске. В этом случае возможны следующие ситуации:

1) освобождаемый участок находится между свободными и в результате образуется большой свободный участок (рис. 54),

2) освобождаемый участок находится между занятым и свободным участками (рис. 55),

3) освобождаемый участок находится между занятыми участками (рис. 56).

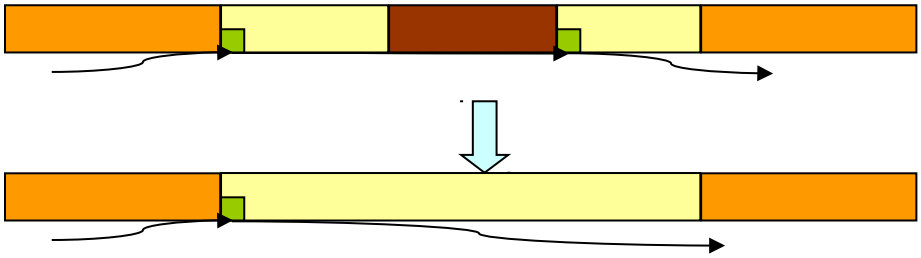


Рис. 54 Освобождение участка находящегося между двумя свободными

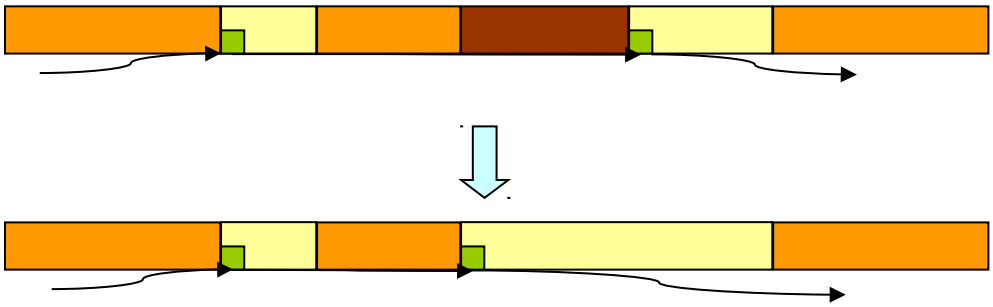


Рис. 55 Освобождение участка находящегося между свободным и занятым

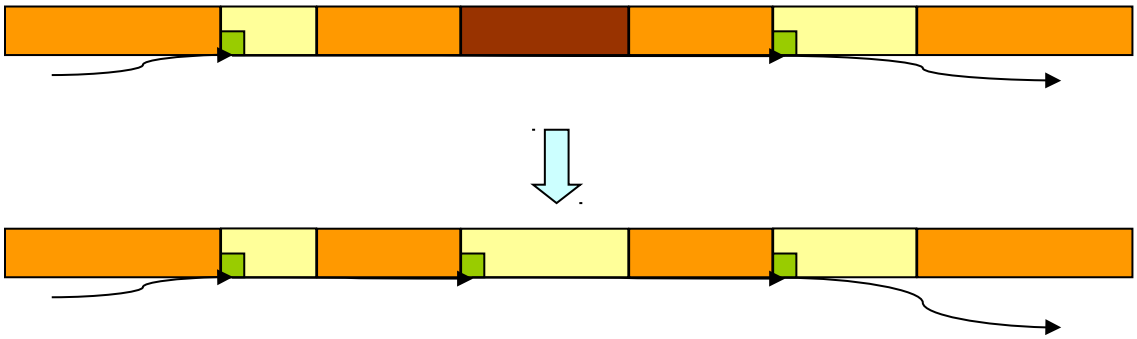


Рис. 56 Освобождение участка находящегося между двумя занятыми

Достоинства и недостатки динамического распределения памяти.

К достоинствам распределения памяти, управляемой динамическими разделами, относится следующее:

1) обеспечивает мультипрограммный режим работы ОС,

2) более эффективное использование памяти по сравнению со статическим распределением,

3) алгоритмы достаточно просты,

4) аппаратные затраты невелики.

Недостатки динамического распределения памяти:

1) основной недостаток – фрагментация памяти,

2) объем памяти, требуемый задаче, может быть настолько большим, что ее будет недостаточно, даже в отсутствии фрагментации,

3) в память загружаются те части кода, которые используются весьма редко, например, функции обработки ошибок. Это характерно для всех методов, использующих загрузку всего логического адресного пространства в физическую память.

Проблема фрагментации при распределении динамическими разделами

Под фрагментацией понимают такое состояние памяти, при котором образуется много мелких свободных участков и создается ситуация, когда не существует непрерывного участка памяти, удовлетворяющего запросу задачи, но общий объем свободной памяти достаточно большой для загрузки задачи в ОП.

Различают внешнюю и внутреннею фрагментации.

Внешняя фрагментация создается в ОП между задачами.

Внутренняя фрагментация образуется в области задачи. Дело в том, что в область задачи загружаются кодовые сегменты, сегменты данных, образуется стек и куча, в некоторых системах в эту область грузятся системные таблицы задачи, методы доступа. Средства борьбы с внутренней фрагментацией располагаются в библиотечных функциях, которые работают с кучей, либо используется загрузка с противоположных концов раздела системной информации и сегментов прикладной задачи.

Для борьбы с внешней фрагментацией разрабатывались специальные методы управления ОП. Рассмотрим два из них: распределение перемещаемыми разделами и использование страничной памяти.

Распределение перемещаемыми разделами памяти.

Суть этого метода состоит в том, что все занятые разделы смещаются в одну область и образуется одна непрерывная область свободной памяти (рис. 57).

Однако в программах существуют адресно-зависимые элементы, чьи значения зависят от места расположения в ОП. При перемещении программного кода задач эти адреса должны быть откорректированы. Примерами таких адресно-зависимых элементов являются:

1. значения базовых регистров,
2. команды, содержащие прямые адреса памяти,
3. адреса в списках параметров,
4. указатели в структурах данных.

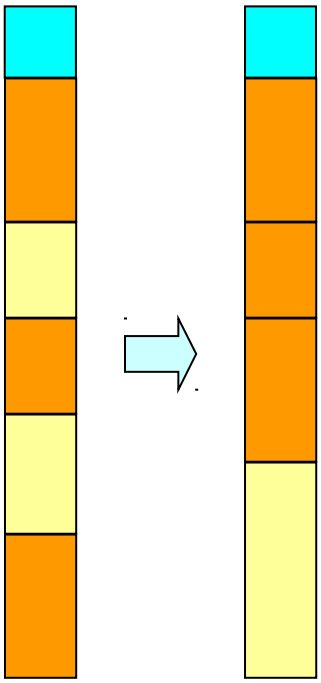


Рис. 57 Перемещение разделов в памяти

Замена значений адресно-зависимых элементов является достаточно трудоемкой операцией. Для этого необходимо хранить информацию об адресно-зависимых элементах, просматривать коды задач и корректировать их значения.

Поэтому количество типов адресно-зависимых элементов было сокращено и остались лишь значения базовых регистров. Это можно было осуществить, если сделать все адреса в области памяти задачи относительными. Тогда при перемещении задачи можно изменить только значения базовых регистров.

Такой метод использовался в системах DEC PDP10, UNIVAC 1108, HoneyWell 6000. Однако перемещение кода также является длительной операцией.

Достоинством метода является ликвидация фрагментации между задачами, что позволяет лучше использовать память, увеличить уровень мультипрограммирования и увеличить производительность системы.

Недостатками этого метода являются следующие:

1) перекомпоновка памяти требует существенных накладных расходов,

2) хотя фрагментация исключается, но часть памяти теряется, так как даже после перемещения освободившаяся память может быть недостаточна для загрузки адресного пространства задачи,

3) память занимают те части кода, которые используются редко.

Управление памятью, организованной страницами.

Идея отказа от загрузки логического адресного пространства задачи в непрерывный участок физической памяти эксплуатировалась достаточно давно, но свою завершенность она обрела с созданием страничной памяти. Все логическое адресное пространство делилось на логические страницы – участки адресов одинаковой длины. Физическая память также делилась на физические страницы. Размер логической страницы равен размеру физической страницы.

Этот метод требовал и существенной аппаратной поддержки в виде механизма страничного преобразования адресов. Адрес представляется в этом случае в виде двух полей: поля номера страницы и поля смещения на страни-

це (рис. 60). Для каждой задачи, загруженной в ОП, создается Таблица страниц задачи. Каждый элемент таблицы со смещением P соответствует логической странице с номером P. Элемент страницы содержит базовый адрес физической страницы, в которой размещена логическая страница с номером P. Для того, чтобы получить действительный адрес для заданного программного адреса (P,D) аппаратура процессора по номеру страницы в поле адреса выбирает элемент таблицы и суммирует базовый адрес физической страницы, находящийся в этом элементе со смещением из поля смещения адреса.

Кроме того, необходимо учитывать свободные и занятые физические страницы ОП. Это можно реализовать с помощью битового вектора, где каждый бит соответствует физической странице.

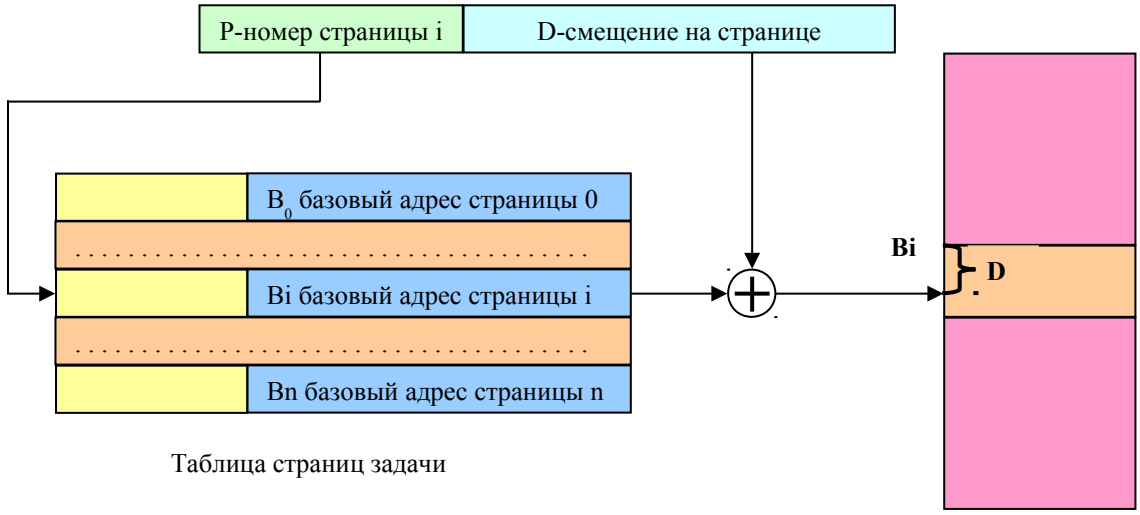


Рис. 58 Схема страничной адресации

Алгоритм работает следующим образом:

Отыскивается необходимое количество свободных физических страниц.

ЕСЛИ число свободных страниц меньше числа логических страниц задачи, ТО задача переводится в состояние ожидания.

ЕСЛИ свободных физических страниц достаточно, ТО логические страницы размещаются в ОП и формируется таблица страниц задачи. Физические страницы помечаются как занятые.

Достоинства метода:

1) исключается внешняя фрагментация,

2) отсутствуют накладные расходы, связанные с перекомпоновкой разделов.

Недостатки метода:

1) остается внутренняя фрагментация, поскольку логическое адресное пространство не кратно размеру страницы и остается незаполненная последняя страница,

2) часть физических страниц остается нераспределенной, так как после загрузки нескольких задач оставшаяся часть свободных физических страниц не удовлетворяет требованиям запросов других задач,

3) в памяти размещается часть программного кода, которая используется крайне редко, поскольку загружается все логическое пространство задачи.

7.3 Управление виртуальной памятью

В случае не страничной организации памяти в ОП загружалось все логическое адресное пространство задачи. Однако нет необходимости иметь весь программный код в ОП, так как не все функции используются при решении конкретной задачи, существуют функции, которые используются крайне редко. Если загружать только ту часть кода, которая необходима в данный момент, то использование памяти будет более эффективным, увеличиться уровень мультипрограммирования, повышается производительность системы.

Идея загрузки только части адресного пространства задачи предполагает, что другая часть адресного пространства хранится во внешней памяти. Таким образом, требуется двухуровневая физическая память и основная память, в которой выполняются задачи, состоит из оперативной памяти и внешней памяти, в качестве которой используется память на диске. Для функций управляющей программы, системных и прикладных задач ОП выступает как виртуальная память большого размера. Функции ядра, которые реализуют управление памятью в этом случае и образуют тот слой программного обеспечения, который обеспечивает виртуализацию памяти.

Существуют следующие способы управления виртуальной памятью:

1) управление виртуальной страничной памятью по запросам,

2) управление сегментной виртуальной памятью,

3) управление виртуальной сегментно-страничной памятью.

7.3.1 Управление виртуальной страничной памятью по запросам

Этот способ управления памятью очень популярен. Он использовался в системах

MULTICS;

IBM370 VS1, VS2, MVS, VM370;

UNIVAC70/46 VMOS;

MS WINDOWS;

Unix, Linux.

Физическая память – двухуровневая, делится на физические страницы (рис.59). Адресное пространство задачи также делится на логические страницы. Размеры логической и физической страниц совпадают. Реально использовались размеры страниц от 256 байт до 4 Кбайт.

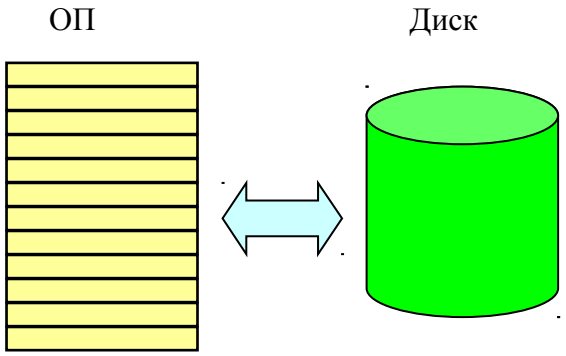


Рис. 59 Обмен страницами ОП и внешней памятью

При выполнении задачи только часть страниц находится в ОП. Остальные страницы находятся во внешней памяти. Если при выполнении программного кода на странице в ОП происходит обращение к программному коду на странице во внешней памяти, то возникает страничное прерывание. Это прерывание происходит во время выполнения команды, при вычислении адреса операнда. Управление по этому прерыванию передается функциям ядра, которые находят нужную страницу во внешней памяти, отыскивают свободную страницу в ОП и подкачивают туда найденную страницу из внешней памяти. Затем повторяется прерванная команда. Если в ОП нет свободной страницы, то функции управления виртуальной страничной памятью выполняют замещение некоторой страницы в ОП. Для этого отыскивается в соответствии с алгоритмом замещения удаляемая страница и на ее место загружается страница из внешней памяти.

Таким образом, при вычислении действительного адреса может возникнуть обмен между ОП и внешней памятью. Поэтому организации внешней памяти для свопинга страниц должно уделяться определенное внимание. В первых ОС с виртуальной страничной памятью использовались специально разработанные внешние накопители информации, но с появлением быстродействующих и надежных жестких дисков стали использовать стандартные устройства компьютера. Однако в различных ОС организация области свопинга различна. Так в ОС Unix, Linux во время установки ОС на компьютер выделяется специальный раздел на диске, а в ОС MS WINDOWS используется системный файл pagefile.sys, который располагается на диске С. Конечно его можно переопределить на другой диск, а также задать минимальный и максимальный размеры, которые должны быть кратны 8. Под минимальный

размер система выделит определенное количество смежных блоков памяти, а в случае необходимости будет выделять дополнительные блоки до максимального размера. Разумеется, в этом случае файл будет фрагментирован, что увеличивает время доступа к внешним страницам. Чтобы смягчить этот недостаток можно создать раздел в виде логического диска размером 1.5-2 Гбайта с именем отличным от “C”. Диск C многие программы используют по умолчанию. А можно просто изменить минимальный размер файла, сделав его равным максимальному.

Разумеется, такая организация управления памятью требует усложнения аппаратуры процессора, системных таблиц и алгоритмов функций управления.

Аппаратура процессора должна поддерживать механизм страничного преобразования адресов и механизм страничного прерывания.

Учет состояния памяти.

Необходимы следующие системные таблицы:

Таблица физических страниц – одна на систему. В таблице существует запись для каждой страницы в ОП. В записи отмечается состояние физической страницы. Возможны следующие состояния:

1) физическая страница свободна и в нее может быть загружена страница из внешней памяти,

2) физическая страница занята,

3) физическая страница находится в транзитном состоянии.

Транзитное состояние физической страницы необходимо для того, чтобы предотвратить конфликты при выборе страницы. Предположим, что некоторая страница свободна и выбрана для загрузки в нее логической страницы задачи из внешней памяти. Операция обмена страниц достаточно длительная операция по сравнению со скоростью процессора, поэтому сделать ее непрерываемой невозможно. Это существенно увеличит накладные расходы, работа управляющей программы будет занимать слишком много времени. Если операция обмена страницами прерываема, то процессор на это время будет занят выполнением другой задачи и возможно для этой задачи также придется загружать страницу из внешней памяти. Состояние памяти к этому моменту не измениться, поэтому для другой задачи будет выбрана та же свободная страница.

Предположим, что страница занята и выбрана алгоритмом замещения для удаления, а на место этой страницы будет загружена страница из внешней памяти. В этом случае возникает такая же конфликтная ситуация выбора

одной и той же физической страницы для двух задач, как и в предыдущем случае.

Чтобы избежать этой конфликтной ситуации, необходимо ввести транзитное состояние, которое будет присваиваться физической странице, когда страница выбрана для загрузки как свободная, или выбрана для выгрузки и последующей загрузки внешней страницы как занятая. Алгоритмы выбора страниц и замещения не рассматривают страницы в транзитном состоянии.

Также в записи таблицы физических страниц находится бит обращения к странице, который устанавливается, если к странице было обращение. Это необходимо для реализации аппроксимации алгоритма замещения LRU. Для удаления выбирается страница, к которой не было обращения, то есть бит обращения не установлен.

Таблица страниц задачи – одна на задачу. Таким образом, сколько создано в системе задач, столько будет создано и таблиц страниц.

Каждая запись в таблице страниц задачи содержит:

1) бит состояния, который показывает, находится ли страница в ОП или во внешней памяти. Когда при обращении к странице соответствующая запись таблицы загружается на управляющий регистр процессора и бит состояния показывает отсутствие страницы в основной памяти, возникает страничное прерывание.

2) базовый адрес страницы физической памяти, в которую загружена логическая страница.

3) бит использования страницы, который устанавливается, если на страницу была произведена запись. В этом случае, при замещении страницу необходимо сохранить во внешней памяти. Если запись не производилась на страницу, то копия страницы хранится во внешней памяти и операция записи не нужна.

Таблица внешних страниц задачи – одна на задачу. Содержит адреса внешних страниц во внешней памяти.

Планирование запросов на выделение памяти.

Запрос на выделение памяти поступает в момент начальной загрузки адачи или выполнения команды задачи и осуществляется по страничному прерыванию. Количество страниц, выделяемое задаче, зависит от алгоритма управления. Некоторые алгоритмы не загружают задачу, пока в ОП не будет достаточного количества страниц, называемого рабочим множеством страниц задачи.

Выделение памяти.

Если существуют свободные страницы, то они выделяются либо при начальной загрузке задачи в ОП, либо по страничному прерыванию.

Если свободные страницы отсутствуют, то необходимо выгрузить какую-либо страницу во внешнюю память, а на ее место загрузить требуемую страницу. Кандидат на выгрузку определяется в этом случае алгоритмом замещения.

Освобождение памяти.

В случае окончания задачи освобождается вся память, как внешние страницы, так и страницы в ОП. В процессе выполнения, страницы задачи могут быть выгружены во внешнюю память по алгоритму замещения.

Алгоритмы замещения страниц.

В случае возникновения страничного прерывания и отсутствия свободных физических страниц в ОП, выполняется алгоритм замещения, который выбирает страницу для удаления. В результате исследований выяснилось, что от применяемого алгоритма зависит производительность системы.

Число страничных прерываний служит оценкой эффективности работы алгоритмов управления памятью. Действительно, при возникновении страничного прерывания выполняется операция обмена с внешней памятью, которая требует существенных временных затрат. Поэтому лучше будет тот алгоритм, который обеспечивает минимальное число страничных прерываний.

Модель производительности алгоритмов замещения позволяет проанализировать свойства различных алгоритмов. Модель включает в себя:

1) Траекторию страниц P – список страниц, в которых расположены используемые при выполнении программы адреса памяти.

2) Размер ОП M, измеряемый в физических страницах.

3) Алгоритм замещения.

Оценка производительности алгоритма, измеряемая как отношение $f = F/|P|$, где f – функция неудач, F – число страничных прерываний, $|P|$ - число страниц в траектории P.

Естественно задать вопрос: «А существует ли оптимальный алгоритм? То есть такой, который для любой траектории страниц обеспечивал бы минимальное значение функции неудач?»

Оптимальный алгоритм.

Оптимальный алгоритм заключается в том, что для удаления следует выбирать страницу, к которой дольше всего не будет обращения.

Пример применения оптимального алгоритма.

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	4	4	4	4	4	**4**	2	2	2
		3	3	3	3	3	3	3	3	**3**	1	1

			2	1	1	**1**	5	5	5	5	5	5
F	+	+	+	+			+			+	+	

F = 7 | P | =12 f=7/12

Единственным недостатком оптимального алгоритма является невозможность его реализации. Узнать траекторию страниц до выполнения задачи невозможно. Как пишут некоторые авторы, если обладать такими возможностями, то лучше играть на бирже. То есть знал бы прикуп, жил бы в Сочи.

Алгоритм FIFO (first in – first out первый пришел – первым ушел).

По этому алгоритму выбирается страница, которая была загружена в ОП первой из всех присутствующих страниц.

Пример применения алгоритма FIFO.

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	**4**	1	1	**1**	5	5	5	5	5	5
		3	3	**3**	4	4	4	4	**4**	2	2	2
			2	2	**2**	3	3	3	3	**3**	1	1
F	+	+	+	+	+	+	+			+	+	

F=9 | P | =12 f=9/12

Этот алгоритм действует неразумно, поскольку удаляет страницы, которые тут же будут использоваться. Кроме того, он обладает и принципиальным недостатком – аномалией страниц.

Если увеличить объем ОП, то естественно ожидать уменьшения числа страничных прерываний. Однако для алгоритма FIFO можно подобрать пример траектории, для которой при увеличении размера ОП число страничных прерываний возрастает.

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	4	4	**4**	5	5	5	**5**	1	1
		3	3	3	3	3	**3**	4	4	4	**4**	5
			2	2	2	2	2	**2**	3	3	3	3
				1	1	1	1	1	**1**	2	2	2
F	+	+	+	+			+	+	+	+	+	+

F=10 | P | =12 f=10/12

Алгоритм LRU (the least recently used самую давно используемую страницу).

По этому алгоритму выбирается страница для удаления, которая дольше всего не использовалась из всех присутствующих страниц, то есть к этой странице дольше всего не было обращений.

Пример применения алгоритма LRU.

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	1	1	**1**	5	5	**5**	2	2	2
		3	3	**3**	4	4	4	4	4	**4**	1	1
			2	2	**2**	3	3	3	3	3	**3**	5
F	+	+	+	+	+	+	+			+	+	+

F=10 | P | =12 f=10/12

Для примера используемой траектории страниц на начальном участке работа этого алгоритма совпадает с работой алгоритма FIFO, но в дальнейшем алгоритм ведет себя иначе. Результат получился хуже для этой траектории, но в алгоритме LRU отсутствует аномалия страниц.

Если сравнить соответствующие столбцы этих двух таблиц, то можно заметить, что множество страниц столбца i при трех физических страницах включается в множество страниц столбца i при четырех физических страницах. Следовательно, невозможна ситуация, когда для страницы i траектории для варианта с тремя физическими страницами не возникало страничного прерывания, а для той же страницы i траектории с четырьмя физическими страницами страничное прерывание возникло.

P	4	3	2	1	4	3	5	4	3	2	1	5
M	4	4	4	4	4	4	4	4	5	**5**	1	1
		3	3	3	3	3	3	3	4	4	**4**	5
			2	2	2	2	2	2	3	3	3	3
				1	1	**1**	5	5	**1**	2	2	2
F	+	+	+	+			+			+	+	+

F=8 │P│=12 f=8/12

Алгоритм замещения страниц LRU также трудно реализовать в чистом виде. Для того чтобы находить страницу, к которой дольше всего не было обращения, необходимо хранить время последнего обращения к странице. Затем при поиске кандидата на удаление следует сравнивать времена всех страниц. Ясно, что это связано с большими накладными расходами. Учитывая, что страничные прерывания возникают при вычислении действительного адреса в команде, а выполняется в среднем лишь одна пятая часть всех команд, расположенных на странице, то прерывания возникают довольно часто. Поэтому используется в реальных системах некоторая аппроксимация алгоритма LRU.

Бит обращения в таблице физических страниц позволяет отличить страницы, к которым было обращение от страниц, к которым не было обращения за определенный отрезок времени. В начальный момент биты обращений всех страниц устанавливаются в 0. В процессе работы если к странице было обращение, то бит выставляется в 1. Если все биты обращений выставлены в 1, то они сбрасываются в 0. Таким образом, за отрезок времени, пока все биты обращений не выставлены в 1, для удаления выбирается одна из страниц с нулевым битом обращения.

Пробуксовка (thrashing).

В страничных системах наблюдается еще одна неприятная проблема, которая получила название «пробуксовка». Это такое состояние системы, когда

идет интенсивный обмен страницами между ОП и внешней памятью, а процессор простаивает, так как все задачи находятся в состоянии ожидания завершения обменом страниц. Каждая задача ждет, когда же будет закачана необходимая ей страница.

Эта ситуация не зависит от применяемого алгоритма замещения страниц, а связана с неограниченным уровнем мультипрограммирования. Количество задач может быть достаточно велико, а объем страниц, выделяемый для одной задачи, соответственно, небольшим. В этом случае часто происходят обращения к страницам, находящимся во внешней памяти. Это и приводит к экспоненциальному росту страничных прерываний.

Стратегия рабочего множества позволяет преодолеть этот принципиальный недостаток. В основе этой стратегии лежит следующая мысль: если в ОП присутствуют те страницы, к которым задача обращается часто, то обмен страницами будет слабым.

Рабочее множество – это такое множество страниц, которое нужно держать в ОП, чтобы в течение достаточно большого промежутка процессорного времени не произошло страничного прерывания.

Определенное количество страниц выделяется в системе в качестве рабочего множества. Если количество свободных страниц меньше рабочего множества, то задача не загружается в ОП. Это приводит к снижению уровня мультипрограммирования и потерям в использовании памяти, если число свободных страниц меньше, чем рабочее множество страниц какой-либо задачи. Система может корректировать рабочие множества страниц задач, оценивая использование страниц в процессе выполнения. Но это приводит к накладным расходам.

При разработке таких систем руководствуются принципами локальности обращений к памяти в программах. Различают локальность во времени и локальность в пространстве.

Локальность во времени состоит в том, что если к данному слову в памяти произошло обращение, то в ближайшее время к этому же слову в памяти опять произойдет обращение.

Локальность в пространстве состоит в том, что если к данному слову в памяти произошло обращение, то в ближайшее время с большой вероятностью произойдет обращение к соседним словам.

Системы программирования должны стараться повысить уровень локальности при построении загрузочного модуля.

Достоинства метода управления виртуальной страничной памятью по запросам следующие:

1) большая виртуальная память,

2) более эффективное использование ОП, за счет загрузки только части адресного пространства задачи,

3) неограниченное мультипрограммирование.

Недостатки метода:

1) накладные расходы возрастают,

2) пробуксовка ограничивает уровень мультипрограммирования, и ухудшает использование памяти.

Метод управления виртуальной страничной памятью по запросам очень популярен и используется во многих ОС.

7.3.2 Управление виртуальной сегментной памятью по запросам

В ранее рассмотренных методах логическое адресное пространство задачи рассматривалось как односегментное, то есть не разделялась память под программы и данные, не выделялись отдельные структурные единицы данных и кода, такие как функции, массивы, другие структуры данных, объекты классов, то есть те, с которыми имеет дело программист. Эти единицы данных и кода представляются как сегменты. То есть сегмент – это логическая группа информации. Обмен между ОП и внешней памятью осуществляется сегментами. Сегменты имеют переменную длину и это обстоятельство вызывает большие сложности при управлении виртуальной сегментной памятью.

Такой способ управления памятью использовался в OS/2.

Программный адрес при сегментной организации логического адресного пространства имеет вид (S, D), где S – номер сегмента, а D – смещение на сегменте. Сегмент имеет переменную длину, поскольку смещение D занимает определенное число разрядов n в адресе, то размер сегмента может быть не более чем 2^n-1, то есть размер сегмента ограничен.

Организация физической памяти также двухуровневая. Так как сегмент имеет переменную длину, то управление ОП и внешней памятью усложняются.

Аппаратные средства процессора должны поддерживать механизм динамического сегментного преобразования адресов (рис. 60) и защиту сегментов.

Учет состояния памяти.

1) Для каждой задачи в момент ее создания образуется таблица сегментов. Номеру сегмента в программном адресе соответствует элемент таблицы сегментов, который содержит следующие поля:

1. базовый адрес сегмента Bi в ОП,
2. бит состояния, показывающий находится сегмент в ОП или во внешней памяти,
3. бит использования, показывающий производилась ли запись в сегмент,
4. длина сегмента.

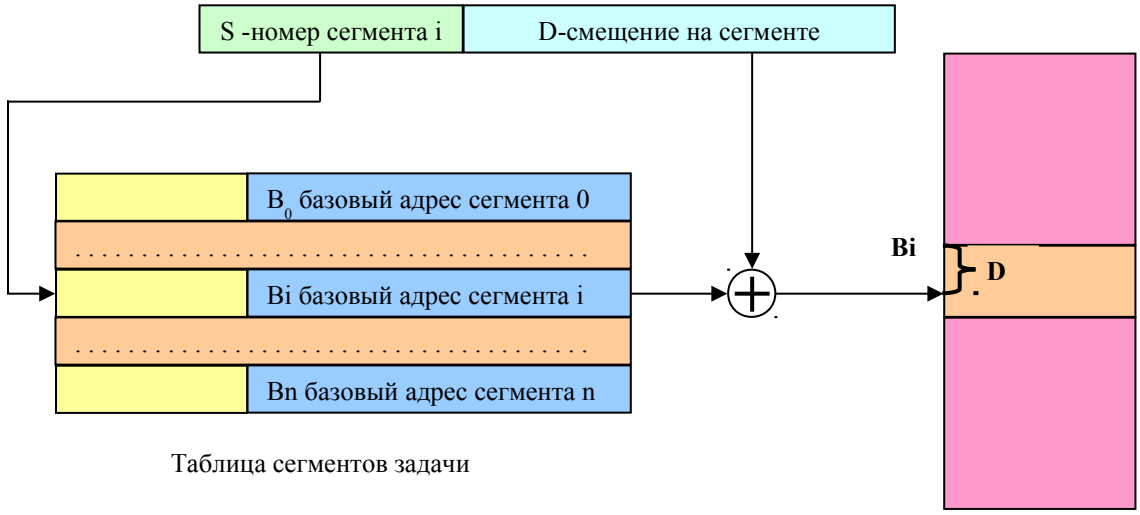


Рис. 60 Схема сегментной адресации

Бит состояния вызывает сегментное прерывание, если сегмент, содержащий адрес, к которому произошло обращение, находится во внешней памяти. Бит использования позволяет не переписывать сегмент во внешнюю память, если в сегмент не происходило записи. Длина сегмента позволяет контролировать адресное пространство сегмента.

2) Учет свободных участков ОП может быть выполнен с помощью списка свободных блоков, который использовался в методах управления ОП, выделяемой динамическими разделами. Начало списка находится в ядре ОС. Поскольку сегменты имеют переменную длину, то возникает аналогичная проблема фрагментации памяти. Методом борьбы с фрагментацией в этом случае является откачка подходящего сегмента во внешнюю память.

3) Учет внешних сегментов задачи также выполняется для каждой задачи и в таблице внешних сегментов содержатся адреса сегментов во внешней памяти.

Планирование запросов на выделение памяти.

Запрос на выделение памяти поступает либо в момент загрузки задачи, либо в момент выполнения задачи и осуществляется по сегментному прерыванию. Для задачи выделяется участок ОП размером, достаточным для загрузки сегмента.

Выделение и освобождение памяти.

Для выделения памяти необходимо найти свободный участок, достаточный для размещения сегмента задачи. Для этого могут быть использованы те же алгоритмы, что и для управления памятью, выделяемой динамическими разделами. Если свободного участка достаточного размера нет, то выбирается сегмент внешней памяти для замещения.

Освобождение памяти осуществляется по окончанию задачи и при замещении сегмента.

Достоинства метода управления виртуальной сегментной памятью состоят в следующем:

1. ликвидация фрагментации,
2. большая виртуальная память,
3. более эффективное использование памяти за счет загрузки только используемых сегментов,
4. понятие сегмента более близко к логической структуре программы, следовательно, это упрощает разработку систем программирования.

Недостатки метода:

1. большие накладные расходы и сложности связанные с управлением ОП и внешней памятью, поскольку сегменты имеют переменную длину,
2. размеры сегмента ограничены,
3. наблюдается явление пробуксовки,
4. большие размеры сегментов препятствуют эффективному использованию памяти.

7.3.3 Управление виртуальной сегментно-страничной памятью

Комбинация методов управления виртуальной сегментной и страничной памятью позволяет устранить основной недостаток сегментной памяти – переменный размер сегментов и сохранить преимущества сегментной организации. Это достигается тем, что логическое адресное пространство состоит из сегментов, а обмен между ОП и внешней памятью осуществляется страницами.

Для того чтобы реализовать этот способ необходимо разбить логическое адресное пространство задачи на сегменты, а каждый сегмент на страницы. Обмен между ОП и внешней памятью осуществляется страницами таким же способом, как и в виртуальной страничной памяти. Поэтому в ОП может быть загружена только часть сегмента задачи. Таким образом, многосегментное адресное пространство задачи отображается на страничную память.

Программный адрес в этом случае состоит из трех полей (S,P,D), где S – номер сегмента задачи, P – номер страницы сегмента и D – смещение на странице. Аппаратные средства процессора по программному адресу динамически вычисляют действительный адрес памяти. Схема динамической переадресации представлена на рис.61. Как видно из этой схемы при вычислении действительного адреса по таблице сегментов задачи находится базовый адрес таблицы страниц сегмента, а затем по таблице страниц сегмента определяется базовый адрес страницы в ОП. Таким образом, накладные расходы возрастают.

Процессор поддерживает два прерывания: сегментное и страничное. Сегментное прерывание возникает, когда таблица страниц данного сегмента отсутствует в памяти, а страничное – когда страница отсутствует в ОП.

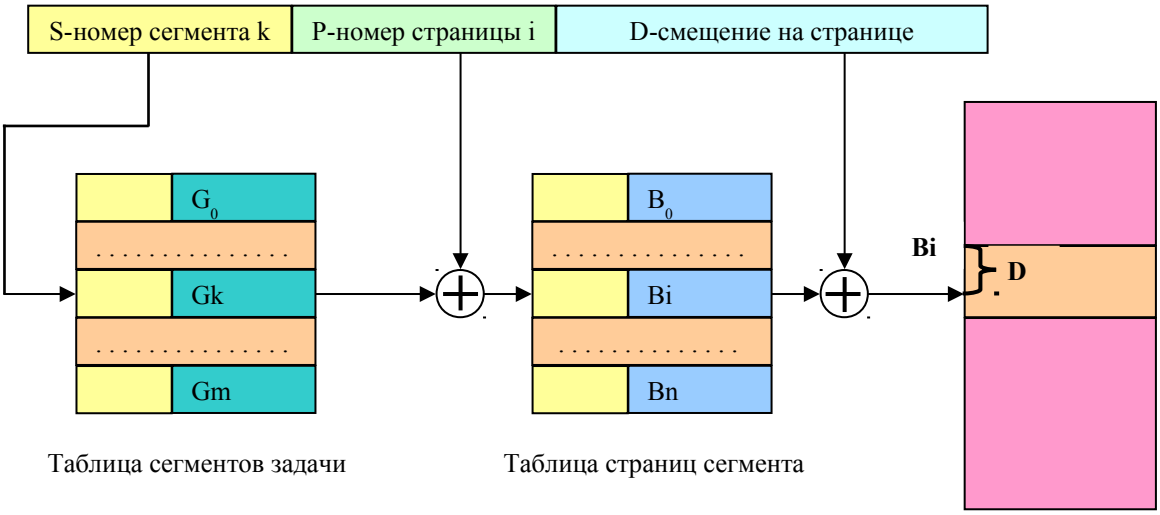


Рис. 61 Схема сегментно-страничной адресации

Учет состояния памяти.

1) Таблица сегментов задачи связывает сегменты и страницы, принадлежащие сегментам. Каждый элемент таблицы содержит базовый адрес таблицы страниц данного сегмента. Кроме того, элемент таблицы сегментов должен содержать бит состояния, который показывает, загружена или нет таблица страниц данного сегмента. Если таблица страниц не загружена в ОП, то возникает сегментное прерывание. Обработка этого прерывания завершается

загрузкой таблицы страниц сегмента. Таблицы страниц сегментов могут также располагаться на страницах и участвовать в страничном обмене.

Кроме того, элемент таблицы сегментов может содержать поля, связанные с использованием таблицы страниц сегмента и полномочиями доступа к сегменту, то есть структура элемента таблицы сегментов аналогична структуре таблицы сегментов при стратегии управления с сегментным распределением, но вся информация относится к таблице страниц сегмента.

2) Таблица страниц сегмента устроена так же, как и таблица страниц при стратегии управления страничной памятью с замещением страниц. Таким образом, в системе могут возникать два типа прерываний при обращении к памяти: страничное и сегментное.

Поскольку обмен на физическом уровне осуществляется страницами, то необходимы таблица физических страниц и таблицы внешних страниц для каждого сегмента.

Планирование запросов при выполнении задачи связано с обработкой страничных прерываний, когда требуемая страница отсутствует, и обработкой сегментных прерываний, когда отсутствует в ОП таблица страниц сегмента. Планирование запросов и выделение и освобождение памяти осуществляется, так же как и в случае стратегии управления виртуальной страничной памятью. Используются те же алгоритмы замещения и стратегия рабочего множества.

Достоинства метода управления виртуальной сегментно-страничной памятью состоят в следующем:

1. большая память, предоставляемая пользователям за счет реализации идеи виртуальной памяти;
2. эффективное использование ОП за счет загрузки только необходимой части адресного пространства;
3. нет ограничений на размер сегмента;
4. отсутствуют недостатки, вызванные использованием сегментов переменной длины;
5. как и в стратегии управления с сегментным распределением, можно поддерживать различные виды доступа к сегментам и защиты сегментов.

Таким образом, основные недостатки стратегии управления с сегментным распределением удалось преодолеть, сохранив преимущества.

Однако недостатки стратегии управления страничной памятью с замещением страниц остались. К ним относится явление "пробуксовки". Другие

недостатки, связанные с усложнением аппаратного обеспечения, затратами памяти под системные таблицы, усложнением программного обеспечения, возросли. Тем не менее, эта стратегия использовалась в нескольких ОС, которые нашли широкое применение. Использование этой стратегии возможно только в больших и мощных компьютерных системах.

Тема 8. УПРАВЛЕНИЕ ПРОЦЕССАМИ И РЕСУРСАМИ В ОС

Операционную систему можно представить как совокупность конкурирующих за ресурсы процессов. Хотя различные ОС исповедуют разные понятия процесса, существуют общие принципы, которые выполняются в ОС разного назначения.

В настоящем изложении обсуждаются эти принципы и особенности, а также рассматриваются примеры из различных ОС. Прежде всего, приводятся определения понятий процесса и ресурса. Рассматриваются алгоритмы диспетчеризации процессов, то есть правил, в соответствии с которыми процессам выделяется процессорное время.

Другой важный вопрос связан с взаимодействием и синхронизацией процессов. Рассматриваются проблемы и методы синхронизации. В различных системах используются различные методы синхронизации. Наибольший набор синхронизирующих примитивов реализован в системе реального времени QNX. В пакетной системе z/OS MVS их гораздо меньше, всего два.

Важный вопрос связан с тупиковыми ситуациями, возникающими при взаимодействии конкурирующих процессов. Обсуждаются вопросы возникновения тупиков, их обнаружение и предотвращение.

8.1 Понятие процесса

Корректное определение понятия «процесс» очень важно, поскольку позволяет понять концепции, заложенные в различные ОС и то, как организована управляющая программа и алгоритмы управления.

Часто понятие процесса отождествляют с программой. Но это приводит к путанице и не позволяет ясно понять работу ОС. Действительно, каждый процесс создается для того, чтобы в его рамках выполнялись программы. С другой стороны, программа в мультипрограммной системе выполняется в рамках некоторого процесса. И все же понятие программы и процесса различны. Программы могут выступать как ресурсы, разделяемые несколькими процессами.

Процесс (задача) это независимая единица работы, участвующая в конкурентной борьбе за ресурсы системы с другими такими работами.

Какую же работу выполняет процесс? Разумеется, ту, для которой и создаются компьютеры. Это работа по преобразованию и переработке информации.

Процесс представляется в ОС некоторым адресным пространством ОП, которая отводится для размещения программ и данных процесса. Также со-

здается Блок Управления Процессом (PCB – process control block). В разных системах он может называться разным образом. В z/OS MVS это блок управления задачей (Task Control Block). В Open System его называют Дескриптором Процесса. PCB создается при создании процесса и размещается в системной области ОП.

Гнезда для PCB могут быть отведены в статическом пуле, как это принято в Unix. В этом случае отыскивается свободное гнездо и там размещается дескриптор процесса. Такой способ более простой и более быстрый. В Linux, z/OS MVS системные блоки размещаются в области системных очередей (SQA). Эта область организована по принципу кучи и все проблемы, связанные с управлением кучей присутствуют при таком подходе. Преимуществом является то, что при недостатке памяти для размещения нужно просто увеличить SQA, а не перекомпилировать ядро.

При завершении процесса PCB уничтожается, освобождается занимаемое процессом адресное пространство, закрываются открытые наборы данных и другие соединения.

8.1.1. Содержимое Блока Управления Процессом

PCB предназначен для хранения системной информации о процессе на время всей жизни процесса в системе. Часть этой информации постоянна и не меняется за время жизни процесса. Другая часть отражает текущее состояние процесса и постоянно изменяется.

Имя и идентификатор процесса

Имя процесса это имя запускаемой в рамках этого процесса первой программы. В некоторых ОС эта программа может подгружать и другие программы. Поэтому эта первая программа, с которой начинается выполнение кода, является головной.

В некоторых системах это имя и идентифицировало процесс в системе. Но проблема заключается в том, что несколько пользователей одновременно захотят выполнить эту программу. Например, компилятор или текстовый редактор. Поскольку идентификация процесса в системе должна быть уникальной, для второго и последующих процессов, использующих одну и ту же программу, имя изменялось путем приписывания номера использования. Например, при использовании компилятора PASCAL в ОС RSX11M строились процессы с именами PASCAL1, PASCAL2 и так далее. Разумеется, такое именование неудобно.

В современных системах перешли на динамическую идентификацию процессов, следуя идеи Unix. Имя процесса так и определяется именем го-

ловной программы, но оно не уникально. Уникальным является идентификатор процесса (PID), который формируется по таймеру или другим способом и представляет беззнаковое целое.

Таким образом, имя и идентификатор процесса характеризуют процесс в системе. Особенно, в силу уникальности широко используется PID для взаимодействия процессов.

Информация о состоянии процессора и полномочиях процесса

PCB должен хранить контекст процесса при блокировании процесса, когда он покидает центральный процессор.

Контекст процесса включает всю информацию необходимую для восстановления процесса и дальнейшего продолжения его работы. В контекст входят счетчик адреса команд, содержимое регистров, регистр флагов и другие данные в зависимости от типа процессора.

Полномочия процесса определяются его типом (системный, пользовательский) и поддерживаются процессором, его состоянием, маскированием прерываний, ключом защиты памяти. Состояние процессора определяется следующими составляющими:

Состояние «супервизор/пользователь» — это состояние определяет для процессора подмножество допустимых команд. В состоянии «супервизор» допустимы все команды и в этом режиме выполняются системные процессы. В режиме «пользователь» непосредственное выполнение ряда команд приводит к аварийному завершению и запрещено в программах процесса. Это команды ввода/вывода непосредственно управляющие устройствами, изменение ключей защиты памяти, слова состояния программы и другие.

Маскирование прерываний также зависит от типа программы. Так в ядре существуют функции супервизора, которые должны быть непрерываемы. Это реализуется маскированием всех прерываний за исключением прерываний от схем контроля. Пользовательские процессы могут маскировать только ограниченное число прерывания. Это прерывания по переполнению, исчезновению порядка.

Ключ защиты памяти является двоичным кодом, который хранится в Слове Состояния Программы (PSW), а также приписывается блокам памяти, распределенным процессу. При обращении к памяти код в PSW и блоке памяти сравнивается. Если код совпадает, то блок принадлежит задаче и операция выполняется. В противном случае происходит программное прерывание по защите памяти. Так происходит для пользовательских процессов. У си-

стемных процессов ключ имеет нулевое значение, что позволяет получать доступ к любому блоку памяти.

Информация об основной памяти процесса

PCB содержит информацию о разделе памяти, распределенной процессу. Это может быть указатель на список блоков управления памятью в нестраничной системе, указатель на таблицу страниц или сегментов в системах с виртуальной памятью.

Информация о ресурсах, распределенных процессу

Это может быть указатель на элемент запроса в списке запросов к повторно-используемому ресурсу. Это может быть список созданных процессом ресурсов. Например элемент в очереди к таймеру или элемент в очереди ENQ/DEQ. Также это могут быть указатели на дескрипторы открытых файлов.

Указатели на родственные процессы

Процесс может создать ряд других процессов. Те в свою очередь также могут создавать процессы. Таким образом образуется дерево родственных процессов. В PCB находится указатель на PCB родственного процесса. Это позволяет из одного процесса получать информацию о родственном процессе. Но это возможно лишь с родственными процессами. Если процессы не родственные, то необходимы специальные системные средства, которые позволяют одному процессу зарегистрироваться в базе данных по имени, а другие процессы используя это имя могут получить PID этого процесса.

Информация о состоянии процесса

Во время своей работы процесс переходит из одного состояния в другое. Основные состояния следующие:

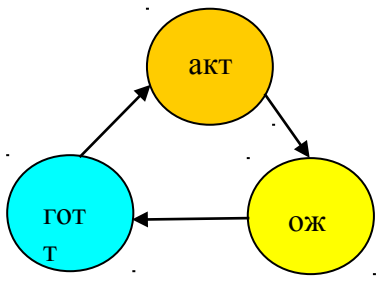


Рис. 62. Диаграмма состояний процесса в пакетной системе

1. активное состояние — программы процесса выполняются на процессоре;
2. состояние ожидания — программы не выполняются на процессоре и процесс ждет завершения событий, например, окончание ввода/вывода;
3. состояние готовности — программы процесса не выполняются на процессоре, но все события уже завершились и процесс стоит в очереди к процессору.

Переходы из состояния в состояние указаны стрелками. Переход из активного состояния в состояние ожидание происходит по инициативе самого активного процесса. Активный процесс запрашивает необходимый по логике его работы ресурс. Это может быть запрос ввода/вывода, например. Если этот ресурс недоступен, то процесс переходит в состояние ожидания.

Процесс находится в состоянии ожидания до завершения ожидаемых событий, завершение ввода/вывода, например. Когда все ожидаемые события завершились, процесс переходит в состояние готовности и помещается в очередь готовых процессов, которые претендуют на занятие процессора.

Представленная диаграмма на рис. 62 отражает диаграмму состояний процессов в пакетной мультипрограммной системе. Дело в том, что в этом случае процесс занимает процессор до тех пор, пока по собственной инициативе его не покинет. Это так называемая «невытесняющая многозадачность».

Другая ситуация складывается в диалоговой системе. В таких системах перед тем как загрузить программы очередного процесса на процессор, функция управляющей программы Диспетчер устанавливает в таймере интервал времени.

Таймер является устройством компьютера, которое работает следующим образом. Каждый такт времени из таймера вычитается определенное значение, определяемое конструкцией таймера. Когда значение в таймере достигнет 0, возникает прерывание от таймера.

Обработчик прерываний управляющей программы анализирует прерывание и передает управление Диспетчеру, который определяет, что интервал времени для выполнения текущего процесса окончен. Диспетчер сохраняет контекст текущего процесса и загружает на процессор программу следующего процесса, для которого также выставляется интервал времени.

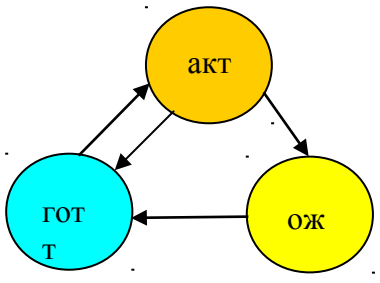


Рис. 63. Диаграмма состояний процесса в диалоговой системе

В литературе для такого способа диспетчеризации иногда используется термин "вытесняющая многозадачность". Хотя реально никто никого не вытесняет. Просто управляющая программа использует аппаратные возможности компьютера и организует такой способ распределения процессорного времени среди процессов в состоянии "готовность".

Таким образом процесс для которого закончился интервал времени переходит в состояние "готовность" (Рис. 63). Он

не ждет никаких событий и нуждается только в очередном интервале процессорного времени.

Диаграммы состояний в разных системах могут быть различными. Также различные диаграммы могут соответствовать различным типам процессов в системе. Но эти различия в основном касаются введением нескольких типов состояний "ожидания". Эти новые состояния являются модификациями трех основных состояний.

Информация о приоритете процесса

Блок управления процессом может содержать приоритет процесса. Как правило это целое число в соответствии с которым процесс занимает свое место в системных очередях. Так в UNIX для системных процессов приоритет является статическим и определяется вторым параметром функции SLEEP. А пользовательские процессы имеют динамический приоритет изменяющийся в зависимости от времени нахождения процесса в основной памяти.

8.1.2. Структура процесса

Следуя идеям UNIX в начальных вариантах системы процесс рассматривается как последовательный виртуальный процесс. Структура такого процесса содержит виртуальное ядро в виде функций, которые исполняются как повторно используемые функции. Это значит, что функции ввода/вывода и код программы выполняются последовательно. Параллельное выполнение ввода/вывода и кода программы на процессоре невозможно.

Виртуальный процесс содержит кодовый сегмент, сегмент данных и динамический сегмент включающий стек и кучу. На Рис.64 представлена структура двух виртуальных процессов.

Каждый из них содержит ядро в виде библиотеки функций, программный код, сегмент данных, стек и кучу. Все эти элементы отображаются на основную память следующим образом.

Виртуальные ядра отображаются на реальное ядро функции которого должны быть или реентерабельными, либо являться критическими секциями для всех процессов и быть непрерываемыми.

Кодовые сегменты процессов могут выполняться одним сегментом, если он реентерабельный. А вот сегменты данных, стек и куча индивидуальны для каждого процесса и для каждого процесса грузятся в ОП собственные сегмент данных, стек и куча.

Следует отметить, что рассмотренная структура с выделением различных по своему функциональному назначению сегментов характерна для

UNIX и UNIX-подобных систем. Она поддерживается процессорами, которые имеют сегментные регистры для базирования разного типа сегментов.

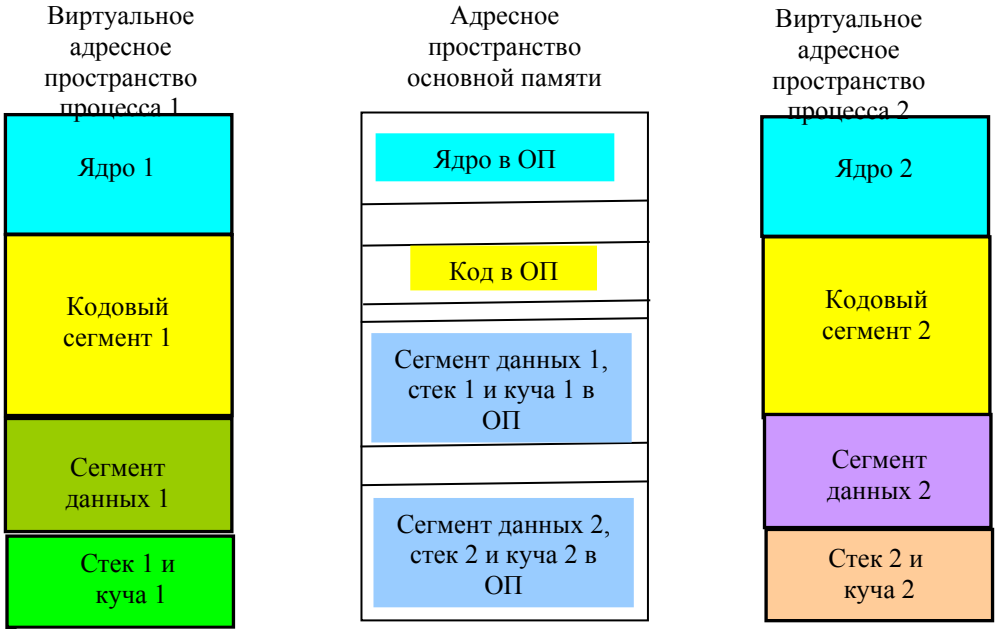


Рис. 64. Структура виртуального процесса и его отображение на основную память

Существует другой подход, когда общие регистры процессора равноправны и нет выделенных сегментных регистров. В этом случае программист сам выбирает регистры для базирования и адресное пространство программного модуля содержит команды и данные. Это характерно для z/OS.

8.1.3. Модель функционирования ОС как совокупность взаимодействующих процессов

ОС можно представить как множество процессов. Следуя UNIX эта модель выглядит следующим образом (Рис.65).

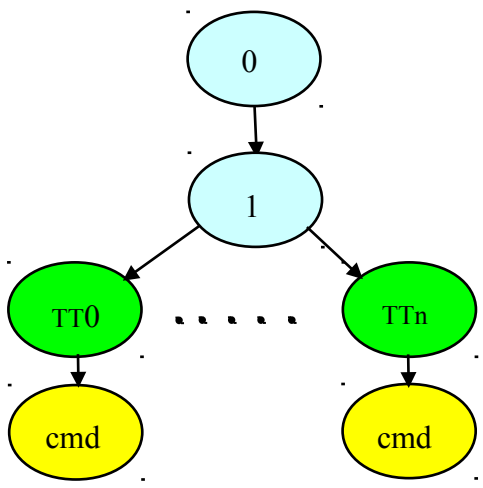


Рис. 65. Схема связей процессов

После загрузки ядра системы создаются следующие процессы. Процесс 0 создается нестандартным образом, то есть не другим процессом, а загруженными программами. Этот процесс отвечает за управление памятью и загрузкой и выгрузкой процессов между областью свопинга на диске и основной памятью.

Процесс 0 создает процесс 1, который является проотцом всех процессов. К этому процессу присоединяются процессы идентифицирующие терминалы, а к ним процессы, запущенные с этих терминалов.

На рис. 65 показаны схема связей процессов. TT0 - TTn процессы, которые создаются при включении (создании) терминалов. cmd - процессы командных процессоров. Они порождают процессы для запускаемых программ.

8.1.4. Процессы и потоки

Первоначальная концепция UNIX предполагала, что все процессы последовательные, а параллелизм возможен только между процессами. Такая модель использовалась для исследования параллелизма и была реализована в UNIX.

Однако развитие аппаратных средств привело к широкому распространению мультипроцессорных систем. Уже процессоры с несколькими ядрами перестали быть экзотикой.

Естественно, что программное обеспечение должно было отвечать этим возможностям. То есть программы процесса должны иметь возможность выполнятся параллельно на разных процессорах. Такая возможность обеспечивается оформлением частей программы в виде потоков (threads).

Поток находится в адресном пространстве процесса, использует данные процесса и кучу. Но каждый поток имеет свой стек. Таким образом все потоки являются просто частями адресного пространства процесса, но могут выполнятся одновременно на разных процессорах. Таким образом процесс потерял такой ресурс как процессорное время. Этот ресурс передан потокам. Система распределяет процессорное время между потоками.

Параллельное выполнение и доступ к общим данным требуют синхронизации потоков. Но функции синхронизации управляющей программы относятся ко всему процессу и переведут все потоки в состояние ожидания, а не один поток. Таким образом синхронизация процессов и синхронизация потоков различаются.

Чтобы решить эту проблему можно сделать отдельный метод синхронизации для потоков на основе программного алгоритма исключения критической секции. Этого недостаточно, поэтому в многопоточных системах используют следующую концепцию. Процесс обладает всеми ресурсам за исключением процессорного времени. Процессорное время распределяется между потоками. Это реализуется в разных системах различным образом. Так в Windows исполняются только потоки и если приложение не имеет потоков, то полагается, что процесс состоит из одного потока. В UNIX все потоки рассматриваются как легкие процессы.

8.1.5. Задачи в z/OS MVS

Другой подход принят в пакетной системе для мэйнфреймов z/OS. В этой системе для каждого выполняющегося задания выделяется адресное пространство и строится управляющий блок адресного пространства (ASCB - address space control block). Все адресные пространства защищены друг от друга также как адресные пространства процессов в UNIX.

В адресном пространстве выполняются задачи, которые определяются шагами задания. Каждая задача подобна процессу может порождать подзадачи, которые в свою очередь также могут порождать свои подзадачи. Для каждой созданной задачи строится блок управления задачей (TCB).

Задача и подзадачи шага задания разделяют адресное пространство. Поэтому другие задачи имеют доступ к задачам в одном адресном пространстве. С этой точки зрения задачи подобны потокам и разделяют время центрального процессора именно задачи.

При таком подходе нет проблемы с разделением прерываний между потоками и процессами. Прерывания относятся к задаче.

8.1.6. Порождение процессов

После загрузки управляющей программы (ядра) при старте системы создается по крайней мере один процесс. Реально в современных системах стартует некоторое множество предопределенных процессов. Среди них обязательно находятся процессы, обеспечивающие связь системы с пользователем. Это командный процессор, графический интерфейс.

Каждый процесс может создать родственный процесс используя функции ядра. Также и командный процессор при вводе командной строки с именем файла загрузочного модуля создает процесс для выполнения этого модуля. При создании процесса выполняются следующие основные действия.

Создается блок управления процессом. Это TCB (task control block) для z/OS или дескриптор процесса для Open system. Размещается этот блок либо в статической памяти зарезервированной в области ядра или динамически строится в специально отведенной области, которая называется областью системных очередей (SQA - system queue area). В UNIX в статической области ядра строится определенное число гнезд для дескрипторов. При создании блока в этом случае отыскивается свободное гнездо, которое заполняется информацией. В остальных системах выделяется память под дескриптор в SQA, которая заполняется информацией о процессе.

Создается идентификатор процесса и заносится в блок управления.

Затем в основную память загружается из файла исполнимый модуль. Когда процессу предоставляется квант времени начинает исполнятся загруженная программа.

В ядре системы существуют функции, которые позволяют запрограммировать создание подпроцесса.

В UNIX появилась интересная функция fork(), которая перешла и в другие системы. Эта функция копирует все адресное пространство в другой вновь созданный процесс. Этот новый процесс имеет свой уникальный идентификатор и дескриптор. В программном коде процесса может быть такой фрагмент:

```
if( id=fork()){
< процесс отца  >
}
else {
< процесс сына>
}
```

В процессе отца функция fork() возвращает отличный от 0 идентификатор созданного процесса, а в созданном процессе возвращаемое функцией значение равно 0. Таким образом в процессах выполняются различные коды.

Также существуют функции, позволяющие запустить в рамках процесса какой-либо другой загрузочный модуль вместо отца.

8.1.7. Уничтожение процессов

Когда программа, выполняющаяся под управлением процесса, заканчивается, то и процесс покидает систему и уничтожается. Уничтожение процесса состоит в следующих действиях.

Освобождается основная память, занятая программами процесса. Уничтожается блок управления процессом. Если он располагается в статической памяти, то ставится отметка о том, что гнездо свободно. Если блок строится в динамической области, то освобождается память.

С уничтожением процесса связана проблема с подпроцессами, порожденными удаляемым процессом. В одном случае их можно также уничтожить вместе с прародителем. Однако существуют ситуации, когда это делать нецелесообразно. Например, пусть запущен процесс печати документа и пользователь решил покинуть систему. В этом случае он выключает терминал и процессы терминала пользователя, командный процессор завершаются. Но процесс печати еще не закончил свою работу и должен продолжать работать. В этом случае он меняет своего прародителя и присоединяется к более старшему процессу, который продолжает работать. Для этой ситуации в

UNIX предусмотрен процесс 1, прародитель всех процессов, который уничтожается при выключении всей системы.

8.2 Понятие ресурса

Ресурсы компьютерной системы по своей физической природе весьма разнообразны. Это процессорное время, основная память, устройства компьютера, файлы, программы, которые одновременно могут использоваться двумя и более процессами. Все эти ресурсы интересны с точки зрения управления их распределением между процессами.

Наиболее важными являются процессорное время и основная память. Эти два ресурса управляются специальным образом. Алгоритмы управления основной памятью были рассмотрены ранее, а управление процессорным временем рассматривается позже.

Способы управления остальными ресурсами зависят от их свойств, а не от физической природы. Поэтому определение и их классификация приводятся далее и не зависят от их природы.

Ресурсы это объекты, которые запрашиваются процессами, используются ими, освобождаются или поглощаются процессами.

Такие ресурсы делятся на разделяемые и неразделяемые.

Неразделяемые ресурсы закрепляются за процессом только раз за время его работы. Такие ресурсы называются ещё монопольно используемыми. Примером такого ресурса является лентопротяжный механизм (ЛПМ). Действительно, устанавливаемая на ЛПМ магнитная лента может использоваться в силу своего устройства только одним процессом. После завершения процесса лента пользователя снимается с ЛПМ и он может использоваться другими процессами.

Разделяемые ресурсы бывают совместно используемыми и повторно используемые разделяемые на последовательной основе.

Совместно используемые ресурсы не требуют какого-либо управления. Примерами таких ресурсов являются общие области памяти используемые двумя и более процессами на чтение. В этом случае информация в общей области не меняется и не зависит от процессов. Если же какой-то процесс пишет в общую область памяти, то этот ресурс перестает быть совместно используемым и должен рассматриваться как повторно используемый. Такая же ситуация возникает и с совместным использованием файлов. Если несколько процессов открывают файл на чтение, то файл является совместно используемым ресурсом. Если часть процессов открывает файл на запись, а другие на чтение, то для процессов открывающих файл на чтение это будет

совместно используемый ресурс, а для процессов открывающих файл на запись это будет повторно используемый ресурс. То есть если какой-то процесс открыл файл на чтение, то другие процессы, открывающие этот файл на чтение могут пользоваться файлом совместно. Но процессы, открывающие этот файл на запись, должны ждать закрытия этого файла. Если какой-то процесс открыл файл на запись, то все остальные процессы должны ждать закрытия файла.

Повторно используемые ресурсы запрашиваются процессом, используются им и затем освобождается. Эта последовательность выполняется несколько раз. Например, процесс ввода читает блок информации с диска и помещает его в буфер. Затем из блока выделяются записи и предоставляются программе. Когда все записи в блоке исчерпаны, процесс ввода читает следующий блок файла и эти шаги действия повторяются. Для того чтобы прочитать блок с диска, необходимо получить доступ к диску. Поскольку диск может в этот момент обслуживать другой запрос на ввод/вывод, то другие процессы должны подождать, когда диск освободится. Таким образом, если диск не является PAV (parallel access volume), то он может в каждый момент обрабатывать только один запрос. Поэтому процессы могут получать доступ к такому ресурсу последовательно. Управление доступом к такому ресурсу требует создания определенных структур данных (рис. 66), которые идентифицируют ресурс, запрос на выполнение операции и программу, управляющую очередью запросов.

Ресурс в системе идентифицируется блоком управления ресурсом. Так для идентификации устройств в системах широко распространен блок управления устройством UCB (unit control block). Управление доступом к повторно используемым ресурсам осуществляется с помощью некоторой программы ядра, которая является менеджером такого ресурса.

Менеджер повторно используемого ресурса имеет два входа. Один вход служит для обработки запроса на доступ к ресурсу, второй вход получает управление, когда требуется освободить ресурс. Он управляет очередью запросов (рис. 66).

Программа процесса при запросе ресурса выдает макрокоманду, которая вызывает прерывание, и управление передается менеджеру ресурса на вход Запрос ресурса. По этому входу строится элемент очереди запросов RQE (element queue request). Этот элемент ставится в очередь и если ресурс занят, то соответствующий процесс переводится в состояние ожидания. Для этого используется ссылка на блок управления процессом PCB.

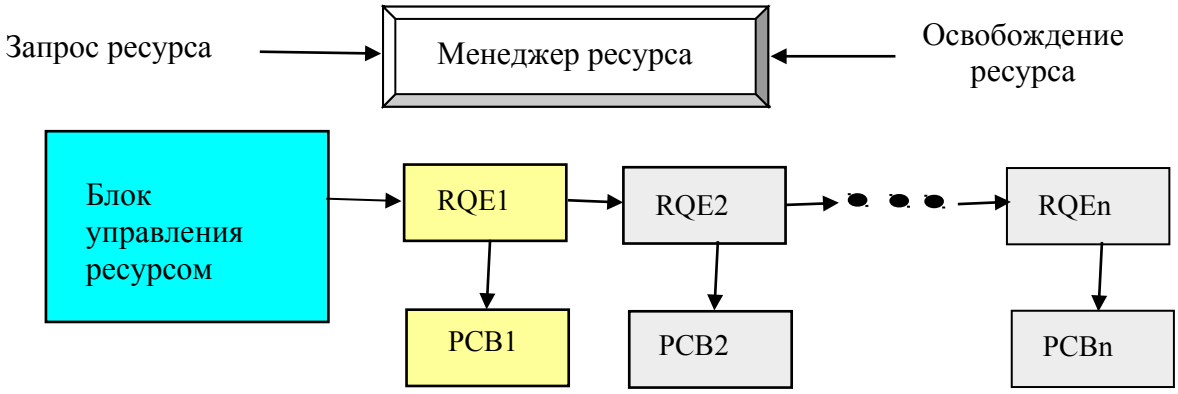


Рис. 66. Управление повторно используемым ресурсом

Процесс владеет ресурсом, если его элемент запроса стоит первым в очереди. Когда процесс, владеющий ресурсом, освобождает его, то происходит прерывание и управление передается на вход Освобождение ресурса. По этому входу менеджер ресурса удаляет RQE этого процесса из очереди и переводит следующий процесс в состояние Готовность, а его элемент запроса стоит первым в очереди.

В качестве примера можно привести физическую систему ввода/вывода, которую реализует функция ядра EXCP (execute channel program) в z/OS.

В отличии от ресурсов, которые все время существуют в системе и могут освобождаться и затем запрашиваться процессами снова, есть ресурсы, которые создаются одними и потребляются другими процессами. Потребляемые ресурсы существуют только определенное время в системе. Таким свойством обладают сообщения, события.

8.3 Планирование процессов

Важнейшим ресурсом компьютерной системы является время центрального процессора. Любое выполнение программы, в том числе и функций ядра, требует занятие процессора на определенное время. Разумеется выполнение системных функций является накладными расходами и должно быть как можно меньше. Поэтому при их реализации используются простые и быстродействующие алгоритмы. История развития ОС знает случаи создания систем, использующих сложные алгоритмы, но эти системы не нашли применения и были скорее исследовательскими. Примером такой системы является разработанная в Массачусетском Технологическом институте система MULTICS.

Распределение процессорного времени между процессами называется планирование процессов. Планирование процесса начинается в момент его перехода из состояния Ожидания в состояние Готовность.

Планирование состоит из следующих шагов.

Во-первых, необходимо назначить процессу приоритет, который определит его место в очереди готовых процессов, или номер очереди готовых процессов, если используется не система приоритетов, а приоритетные очереди.

Во-вторых, происходит непосредственно постановка процесса в очередь готовых процессов.

В-третьих, программа ядра Диспетчер в соответствии с алгоритмом диспетчеризации берет процессы из очереди и предоставляет им процессорное время. Для этого необходимо сохранить контекст предыдущего процесса, то есть запомнить содержимое счетчика адреса команд, содержимое регистров. Запоминание осуществляется в таблицах, связанных с PCB. Затем загружается содержимое регистров нового процесса и его счетчик адреса команд. В результате начинается выполнение программы с прерванного места.

8.3.1 Диспетчеризация процессов

В зависимости от типа системы целесообразно использовать определенные дисциплины диспетчеризации. Для пакетных систем более подходят просто очереди, поскольку время реакции не существенно. А вот для диалоговых систем необходимы дисциплины обеспечивающие разделение времени на кванты. Для систем реального времени требуется дисциплина, позволяющая предварительно оценить масштаб времени каждого процесса.

8.3.1.1 Дисциплины FIFO

Дисциплина "Первый пришел - первый ушел" (First in - first out) это просто очередь. Процесс, который переходит из состояния Ожидания в состояние Готовность ставится в конец очереди, а первый процесс в очереди загружается на процессор при его освобождении.

Часто используется очередь с приоритетами. Приоритеты бывают статические и динамические. Статический приоритет не изменяется все время существования процесса, а динамический корректируется за время жизни процесса.

В качестве примера можно привести диспетчеризацию процессов в UNIX. Системные процессы имеют статический приоритет. Он задается вторым параметром функции ядра sleep(ind,pri). Эта функция переводит процесс в состояние Ожидания. Другая функция wakeup(ind) пробуждает процесс и переводит его в состояние Готовность с приоритетом pri. Следует заметить, что эти функции используются только в функциях ядра, когда процесс исполняется в системной фазе. Приоритет пользовательских процессов меняется динамически и увеличивается по мере нахождения процесса в основной па-

мяти. Следует заметить, что чем больше значение приоритета тем он хуже. Приоритет меняется в соответствии со следующим соотношением.

p_pri = PUSER + p_nice - NZERO + p_cpu/16

Поле p_pri в структуре дескриптора процесса в момент загрузки в основную память имеет значение PUSER поскольку поле p_nice равно NZERO, а поле p_cpu равно 0. В каждом такте передиспетчеризации поле p_cpu увеличивается на 1. Таким образом через каждые 16 тактов поле p_pri будет увеличиваться на 1. Когда приоритет процесса будет больше приоритета какого-либо процесса в области свопинга, то этот процесс будет выгружен в область свопинга, а процесс с меньшим приоритетом будет загружен в основную память. Таким образом происходит смена процессов поскольку UNIX является диалоговой системой.

8.3.1.2 Дисциплина равномерного циклического квантования

Использование очереди FIFO с динамическими приоритетами и передиспетчеризации процессов обеспечивают квантование времени необходимое для диалоговых систем, но квант времени занятия процессора не задается явным образом.

В других дисциплинах для диалоговых систем он задается явно. Простейшей такой дисциплиной является равномерное циклическое квантование RR (Round Robin). Каждый процесс получает квант времени Δt для занятия процессора.

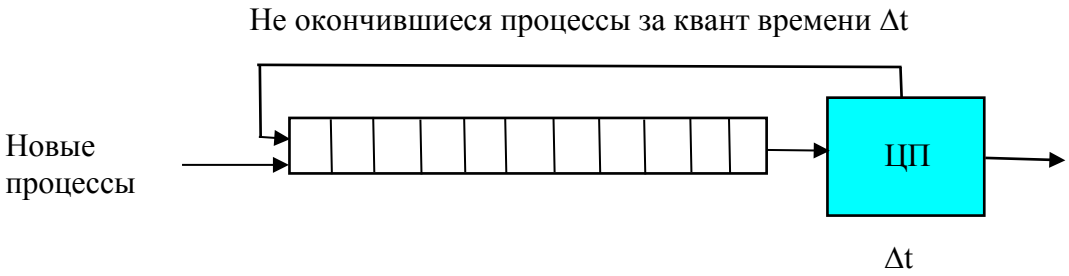


Рис. 67. Равномерное циклическое квантование

На рис. 67 показана схема работы по этой дисциплине. Новые процессы, переходящие из состояния Ожидания в состояние Готовность встают в очередь к ЦП. Контекст очередного процесса загружается на процессор и в таймер загружается интервал времени Δt. Программы процесса исполняются на ЦП. Процесс может покинуть процессор до окончания кванта времени перейдя в состояние Ожидания запросив ввод/вывод, например, или вообще закончится. Если процесс не покинул ЦП, а квант времени закончился, то процесс переводится в конец очереди.

У этой дисциплины существует две модификации.

Модификация Коффмана состоит в добавлении дополнительного кванта процессам в очереди при появлении нового процесса. Это увеличивает вероятность того, что большинство процессов, стоявших в очереди, покинут процессор. Другая модификация состоит в назначении новой заявке столько квантов сколько процессов стоит в очереди. Это также увеличивает вероятность того, что процессы будут покидать процессор. Эти модификации стремятся не позволить очереди готовых процессов расти беспредельно.

8.3.1.3 Дисциплины с обратной связью

Дисциплины с обратной связью позволяют учитывать использование предыдущих квантов процессом. Если процесс не уложился в один квант времени и ему требуются ещё кванты, то он наказывается и будет выполняться после быстрых процессов. Это позволяет реализовать базовую концепцию, согласно которой быстрые процессы пропускаются вперед. Быстрые процессы много работают с внешними устройствами и соответственно мало используют ЦП. Это работа с файлами и базами данных, диалоговые процессы.

Такие дисциплины позволяют разделять процессы на счетные, которые большой процент времени используют ЦП, и диалоговые, использующие процессор сравнительно мало. Они используют несколько очередей. Проштрафившиеся процессы переводятся в старшие очереди, которые обрабатываются, когда в младших очередях нет процессов.

На рис.68 представлена схема работы дисциплины диспетчеризации FBN (Feed back с N очередями). Новые процессы поступают в очередь n=1. Если процесс не покинул процессор за квант времени Δt, то он переходит в очередь n=n+1. Таким образом, чем больше квантов времени потребуется процессу, тем с большим номером очереди будет помещен процесс.

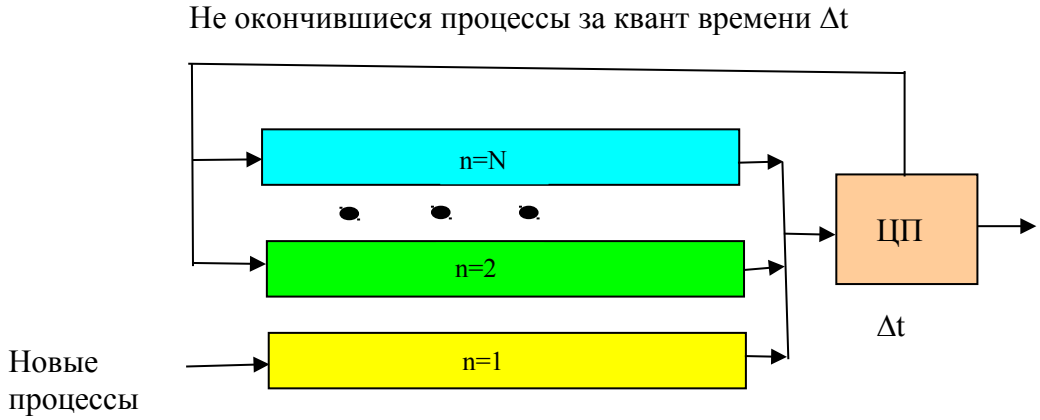


Рис. 68. FB_N дисциплина диспетчеризации

Очереди обслуживаются в порядке возрастания номеров. Сначала обслуживается очередь с новыми процессами n=1. Если эта очередь пуста, то берутся процессы из старшей непустой очереди.

8.3.1.4 Дисциплины диспетчеризации реального времени

ОС реального времени характеризуются фиксированным числом функциональных задач. Поэтому временные характеристики этих задач могут быть изучены и известно расписание, в соответствии с которым будут запускаться эти задачи, последовательность их запуска.

Дисциплина диспетчеризации построена следующим образом (Рис. 69). Выделяется определенное число уровней (обычно 16 или 32). Каждому уровню приписываются задачи. Распределение задач по уровням осуществляется при проектировании системы. Можно на высшем уровне разместить задачи, которые считывают входную информацию с датчиков и посылают управляющие сигналы на объект управления. На следующем уровне поставить задачи фильтрующие полученные значения и отбраковывающие ложные. Далее можно расположить задачи, приводящие полученные коды к физическим величинам. На более низких уровнях располагаются задачи оценивающие параметры, выводящие информацию на дисплей оператора.

Диспетчер, реализующий дисциплину диспетчеризации, просматривает задачи, находящиеся на самом верхнем уровне. Задачи, находящиеся в состоянии готовности, запускаются на выполнение. Если таких задач нет, то обрабатывается следующий нижний уровень. Таким образом программа диспетчер осуществляет только запуск программ.

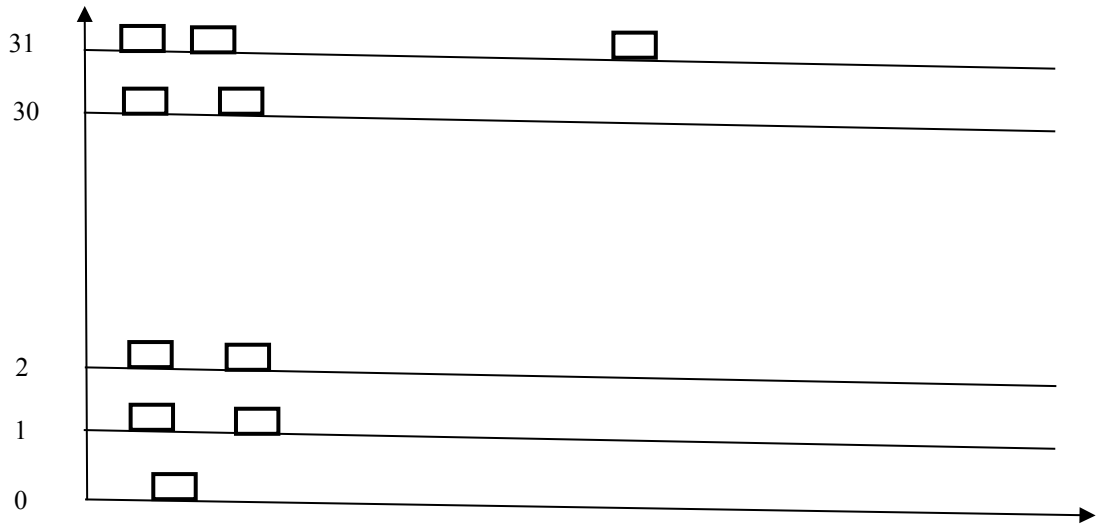


Рис. 69. Дисциплина диспетчеризации реального времени

В ОС реального времени существует некоторое расписание, указывающее в какой момент времени какая программа должна отработать. Это расписание поддерживает специальная задача, переводящая в нужные моменты времени задачи на уровнях в состояние готовности.

8.4 Взаимодействие и синхронизация процессов

Современные приложения разрабатываются как совокупность взаимодействующих параллельно выполняющихся процессов. Эти процессы должны в ходе своей работы синхронизировать свои действия и возможно обмениваться информацией.

8.4.1 Физический и логический параллелизм

Физический параллелизм соответствует такой ситуации в компьютерной системе, когда каждый параллельно выполняющийся процесс выполняется на своем физическом процессоре. В этом случае реально повышается быстродействие.

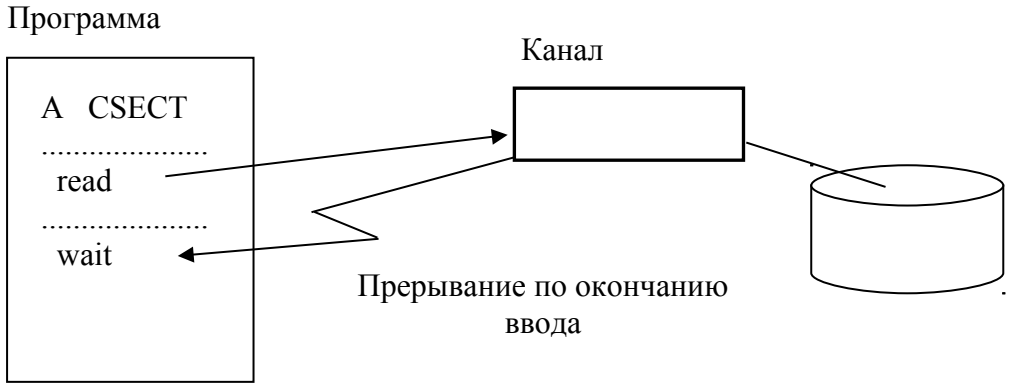


Рис. 70. Физический параллелизм

Впервые такой способ работы был достигнут в системе OS/360. Наряду с ЦП в системе существовали периферийные процессоры ввода/вывода (каналы). Таким образом возможно одновременное выполнение ввола/вывода и работы ЦП для одной программы (рис 70).

Макрокоманда read запускает процесс ввода на периферийном процессоре и одновременно продолжается выполнение программы на ЦП. В точке кода после которой выполнение без введенных данных невозможно стоит макрокоманда wait. Если ввод закончится раньше wait, то выполнение программы продолжится. Если ввод не закончится, то по wait процесс перейдет в состояние Ожидания. По прерыванию по окончанию ввода выполнение программы продолжится.

В современных мультипроцессорных системах используется физический параллелизм для много поточных приложений. В этом случае время ЦП предоставляется потокам, а сам процесс такого ресурса не имеет.

Логический параллелизм не дает увеличение быстродействия поскольку связан с разделением времени процессора между процессами. Более того, на переключение процессов тратится дополнительное время. Функция ядра Диспетчер при переключении процессов запоминает контекст старого процесса на ЦП и загружает контекст нового процесса из очереди готовых процессов. Контекст процесса содержит всю необходимую информацию для продолжения работы. Это содержимое регистров, адрес команды начала выполнения программы, регистр флагов и другая информация.

Пожалуй впервые в широко распространенных ОС логический параллелизм использовался также в OS/360. Задачи планировщика заданий работали именно таким образом. Задача планировщика RDR (reader) вводила текст задания на JCL, обрабатывала его и строила совокупность связанных таблиц, описывающую задание. Эти таблицы ставились в очередь входных работ соответствующего класса и читались из очереди задачей инициатором (INIT). Если очередь пуста, то INIT переходил в состояние Ожидания пока не появится в очереди задание. Эти задачи разделяли время центрального процессора.

8.4.2 Проблемы синхронизации параллельных процессов

Физический и логический параллелизмы с точки зрения синхронизации имеют много общего и используются одни и те же механизмы. Во многих практических случаях встречаются одинаковые ситуации и эти случаи синхронизации формулируются как проблемы. Так выделяют проблемы критической секции, поставщик-потребитель, читатели-писатели, обедающие китайские философы.

8.4.2.1 Проблема критической секции

Простой пример иллюстрирует эту проблему. Пусть два процесса используют общую переменную M, в которой накапливается сумма прохождения в программах процессов определенного участка кода (рис. 71). В начальный момент M=0.

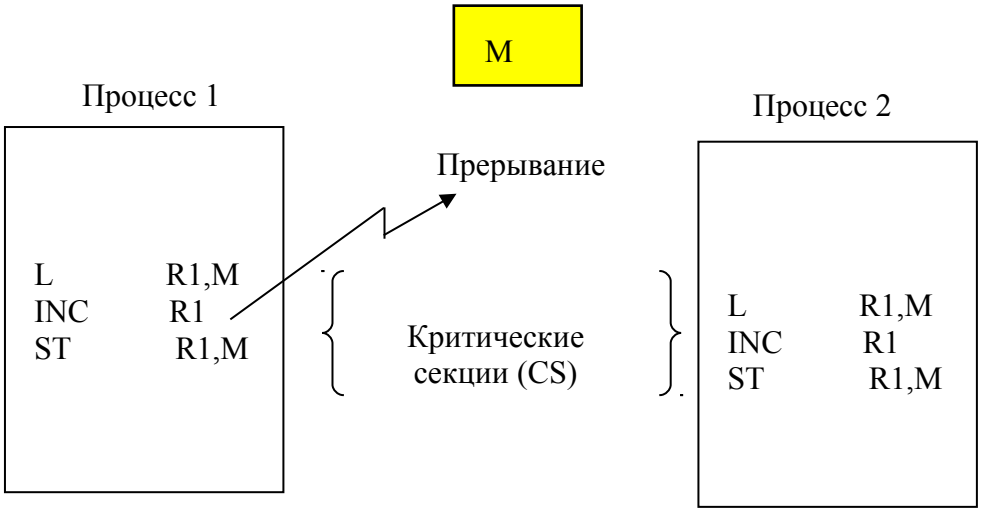


Рис. 71. Проблема критической секции

Предположим, что первым выполняется процесс 1. Программа процесса в определенной части кода читает переменную M в регистр R1, затем прибавляет к регистру 1. Но в этот момент происходит прерывание. Процесс 2 получает процессорное время и его программа проходит часть кода, которая увеличивает значение M. После чего M=1.

Далее процесс 1 получает снова процессорное время и значение R1=1 записывается в переменную M. Таким образом, переменная M имеет значение 1, хотя должно быть 2. Если бы эти участки кода выполнились последовательно, то ошибки бы не произошло.

Критический ресурс это ресурс, который используется несколькими процессами, но разделяется ими на последовательной основе, то есть только один процесс может владеть этим ресурсом. Остальные процессы должны подождать освобождения ресурса.

Часть кода в программах процесса, которые работают с критическим ресурсом называются критическими секциями.

Проблема критической секции состоит во взаимном исключении вхождения двух и более процессов в свои критические секции. Только один процесс может находиться в критической секции. Разумеется, предполагается, что ни один процесс не остается бесконечно долго в своей критической секции и ни один процесс не ждет бесконечно долго вхождения в свою критическую секцию.

Эта проблема широко распространена в практике программирования. В некоторых системах существует специальный механизм MUTEX (mutual exclusion) для решения этой проблемы.

8.4.2.2 Проблема "Поставщик-Потребитель"

Проблема "Поставщик-Потребитель" может быть сформулирована следующим образом. Пусть один процесс получает информацию из внешних источников. Его задача взять из очереди свободный буфер, поместить в него прочитанную информацию и поместить этот буфер в очередь заполненных буферов. Другой процесс берет заполненные буферы, обрабатывает информацию из этого буфера, а освободившийся буфер ставит в очередь свободных буферов (рис. 72).

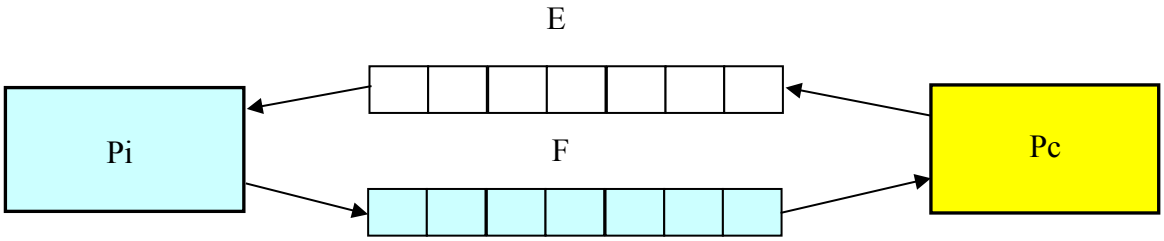


Рис. 72. Проблема "Поставщик - Потребитель"

Процессы синхронизируются на очередях. Если процесс ввода Pi работает быстрее, чем процесс обработки Pc, то очередь пустых буферов E будет исчерпана и процесс Pi перейдет в состояние Ожидания. Он останется в нем до появления пустого буфера. Если процесс Pc работает быстрее, то очередь заполненных буферов будет исчерпана и процесс Pc перейдет в состояние Ожидания. Он будет находится в этом состоянии до появления заполненного буфера.

Пусть E будет переменной, содержащей число буферов в очереди пустых буферов, а F будет переменной, содержащей число буферов в очереди заполненных буферов. Программы процессов можно представить следующим образом (рис. 73).

```
Процесс Pi
PI:if E = 0 then go to PI;
E=E-1;
<взять пустой буфер, ввод,
поставить заполненный буфер>
F=F+1
go to PI;
```

```
Процесс Pc
PC:if F = 0 then go to PC;
F=F-1;
<взять полный буфер, обработка,
поставить пустой буфер>
E=E+1;
go to PC;
```

Рис. 73 Программы процессов Pi и Pc

Конструкция с if проверкой условия и переходом к новой проверке называется активным или занятым ожиданием. Активным поскольку програм-

ма крутится в цикле, проверяя условие. А занятым поскольку процесс занимает весь интервал предоставленного ему процессорного времени. Разумеется, такой способ реально нельзя использовать, но в ядре UNIX нечто подобное используется.

Следует отметить, что переменные E и F являются критическим ресурсом для процессов и по отношению к ним возникает проблема критической секции.

8.4.2.3 Проблема "Читатели-Писатели"

Пусть существуют N процессов Ri, которые открывают файл на чтение и M процессов Wj открывающих файл на запись. Процессы Ri могут открывать файл совместно, а каждый процесс Wj может открыть файл только, если ни один какой-либо процесс не открыл файл. Примером такого взаимодействия процессов может служить файл счетов, который хранит информацию о счетах клиентов в банке. Если клиенты создают запрос с целью узнать состояние счета, то такой запрос выполняет процесс "читатель". Если клиент хочет взять наличные деньги или положить их на счет, то эту операцию осуществляет процесс "писатель".

Синхронизация этих процессов осуществляется следующим образом. Введем общие переменные r и w. Переменная r целого типа содержит число процессов читателей, открывших файл на чтение. Переменная w булевского типа показывает открыт файл на запись или нет. Программы процессов можно представить следующим образом (рис. 74).

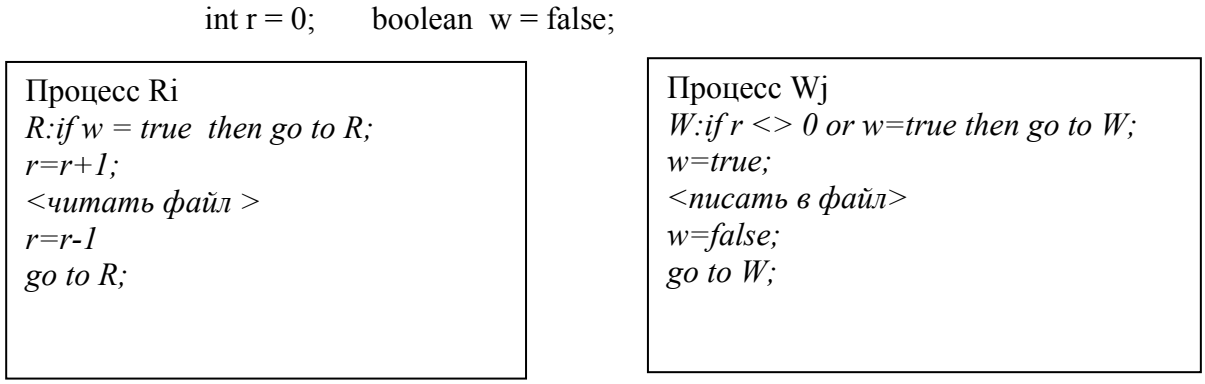


Рис. 74 Программы процессов Ri и Wj

Следует заметить, что общие переменные r и w являются критическим ресурсом и также требуют синхронизации на основе взаимного исключения.

8.4.2.4 Проблема "Обедающие китайские философы"

Известно, что для того чтобы пообедать необходимы две палочки. Проблема формулируется следующим образом.

Существуют n процессов философов и n ресурсов палочек. Каждый процесс имеет доступ только к двум соседним ресурсам. Если эти два ресурса свободны, то философ обедает. Если один из этих ресурсов или оба заняты, то философ ведет беседу. Ситуация для n=3 показана на рис. 75.

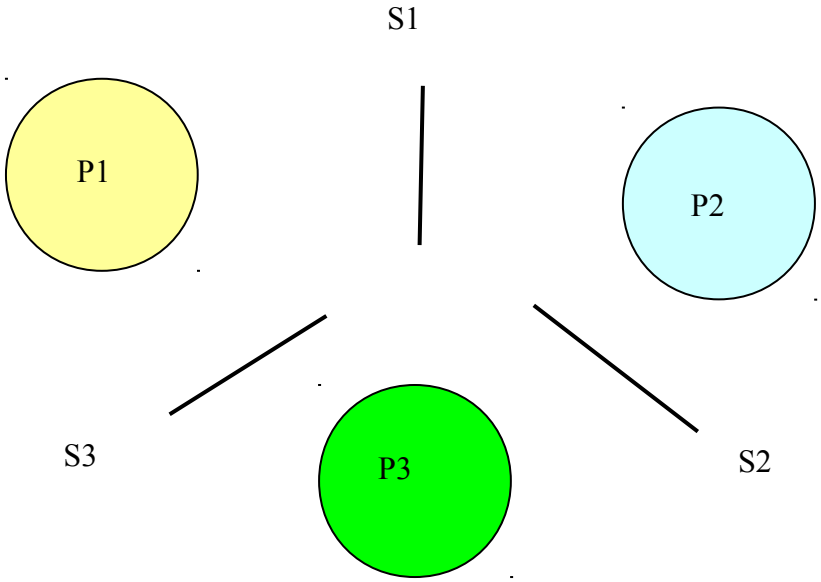


Рис. 75. Проблема обедающих философов.

Процессы философы P1. P2 и P3. Ресурсы палочки S1. S2 и S3. Процесс P1 активен, если ресурсы S1 и S3 были свободны и могут быть заняты процессом. Это же касается процесса P2 и ресурсов S1 и S2, а также процесса P3 и ресурсов S2 и S3. В противном случае процесс ждет освобождения доступных ему ресурсов.

8.4.3 Механизмы синхронизации параллельных процессов в ОС

Различные ОС имеют различные механизмы синхронизации, которые оформляются как функции ядра. Эти функции должны быть непрерываемы также как функции управления основной памятью. Действительно, эти функции изменяют общие для нескольких процессов переменные и следовательно, если не делать их непрерываемыми, то придется синхронизировать внутри этих функций участки кода как в проблеме критической секции.

Следует заметить, что функции синхронизации представлены в виде пары, когда одна функция переводит процесс из Активного состояния в состояние Ожидания (блокирует процесс), а другая переводит процесс из состояния Ожидания в состояние Готовность (разблокирует процесс).

8.4.3.1 Синхронизация функциями wakeup/sleep в ядре UNIX

Функция sleep переводит процесс в системной фазе из состояния Готовность в состояние Ожидания и имеет формат:

sleep(&event,priority);

Первый параметр это адрес события. Под событием понимается некоторая переменная. Это может быть простая переменная, массив, элемент массива, структура и так далее. Второй параметр задает приоритет, который получит процесс после разблокирования.

Функция wakeup переводит процесс в системной фазе из состояния Ожидание в состояние Готовность. Формат функции следующий:

wakeup(&event);

Параметром является адрес события, по которому процесс был заблокирован.

Рассмотрим пример. Пусть один процесс засыпает по событию runout. При этом процесс выдает последовательность команд:

runout++;

sleep(&runout,PSWP);

Другой процесс будит первый процесс последовательностью команд:

if(runout != 0) { runout = 0; wakeup(&runout);};

Оба процесса находятся в системной фазе и первый процесс получает статический приоритет после пробуждения.

8.4.3.2 Активное или занятое ожидание

При рассмотрении проблем синхронизации использовалась конструкция вида:

M: if(cond = 0) goto M;

Этот способ называется активным ожиданием поскольку процесс находится в активном состоянии и весь интервал времени, предоставленный ему на процессоре занимается в цикле проверкой условия. Также этот механизм называют занятым ожиданием поскольку процесс с таким кодом будет занимать весь интервал времени целиком. Такая реализация существенно замедляет работу системы и на практике следует после проверки условия в случае его невыполнения обрывать интервал времени. Как правило в API существуют средства, позволяющие это сделать.

В UNIX существует событие lbolt, которое генерируется системой периодически через определенный интервал времени. Другими словами система периодически выдает wakeup(&lbolt);

Это сделано также для того, чтобы предотвратить ситуацию, когда один процесс выдает wakeup(), а sleep() по этому событию ни одним процессом еще не был выдан.

Реальный код для случая, когда интервал времени обрывается, выглядит следующим образом:

while(cond = 0) sleep(&lbolt,pri);

В цикле проверяется событие. Если оно не наступило, то процесс засыпает. Он проснется с наступлением события lbolt и снова проверит условие наступления события.

8.4.3.3 Семафоры

Семафор это переменная целого типа принимающая значения ≥0. Семафор связан с ресурсом, разделяемым несколькими процессами. Над семафором S определяются две операции.

P(S) выполняет S = S - 1, если S>0. Если S=0, то процесс переводится в состояние Ожидания.

V(S) выполняет S = S + 1.

Решение проблем синхронизации с помощью семафоров приводится ниже.

Решение проблемы критической секции выглядит следующим образом. Пусть заданы n процессов. Каждый имеет критическую секцию CS в коде программы, работающую с общим критическим ресурсом. С этим критическим ресурсом связывается семафор S.

Начальное значение семафора S = 1.

В каждом процессе код с критической секцией выглядит следующим образом.

```
p1: begin .....P(S); CS1; V(S); .... end
 . . . . . .
pi: begin .....P(S); CSi; V(S); .... end
. . . . . .
pn: begin .....P(S); CSn; V(S); .... end
```

Предположим некоторый процесс первым вошел в свою критическую секцию. В этом случае семафор примет значение 0. Поэтому каждый следующий процесс, пытающийся войти в свою критическую секцию будет блокирован. После выхода из критической секции семафор открывается и получает значение 1. Также функция V(S) должна переводить блокированные процессы из состояния Ожидания в состояние Готовность.

Решение проблемы "Поставщик/Потребитель" с помощью семафоров можно сделать следующим образом. Сопоставим очереди пустых буферов семафор E с начальным значением n, а очереди полных буферов семафор F с начальным значением 0.

Тогда процесс ввода будет выглядеть следующим образом:

begin LR: P(E); < ввести запись>; V(F); goto LR; end

Процесс обработки строится аналогичным образом:

begin LC: P(F); < обработать запись запись>; V(E); goto LC; end

Таким образом процесс ввода будет блокирован, когда число пустых буферов станет равным 0, а разблокируется, когда процесс обработки увеличит значение E. Соответственно, процесс обработки блокируется, когда число полных буферов станет равным 0, а разблокируется, если процесс ввода увеличит значение семафора F.

Решение других проблем синхронизации процессов с помощью семафоров рекомендуется сделать читателю.

8.4.3.4 Синхронизация с помощью POST/WAIT в z/OS MVS

Механизм синхронизации POST/WAIT основан на событии. Событие представляется в системе блоком управления событием ECB (event control block). Структура блока показана на рис. 76. Блок представляет собой четырех байтное поле. Первые два бита содержат флаги W и P. Остальная часть блока содержит различную информацию.

Функция WAIT переводит задачу в состояние Ожидания до наступления событий, если эти события не наступили до вызова функции. Если события наступили до вызова функции WAIT, то блокировки задачи не происходит. Функция POST сообщает о наступлении события.

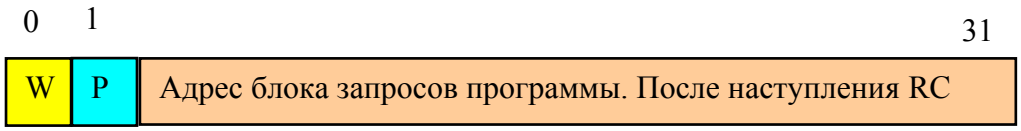


Рис. 76. Блок управление событием ECB

Следующий пример (рис. 77) иллюстрирует работу функций, позволяющих синхронизировать процессы.

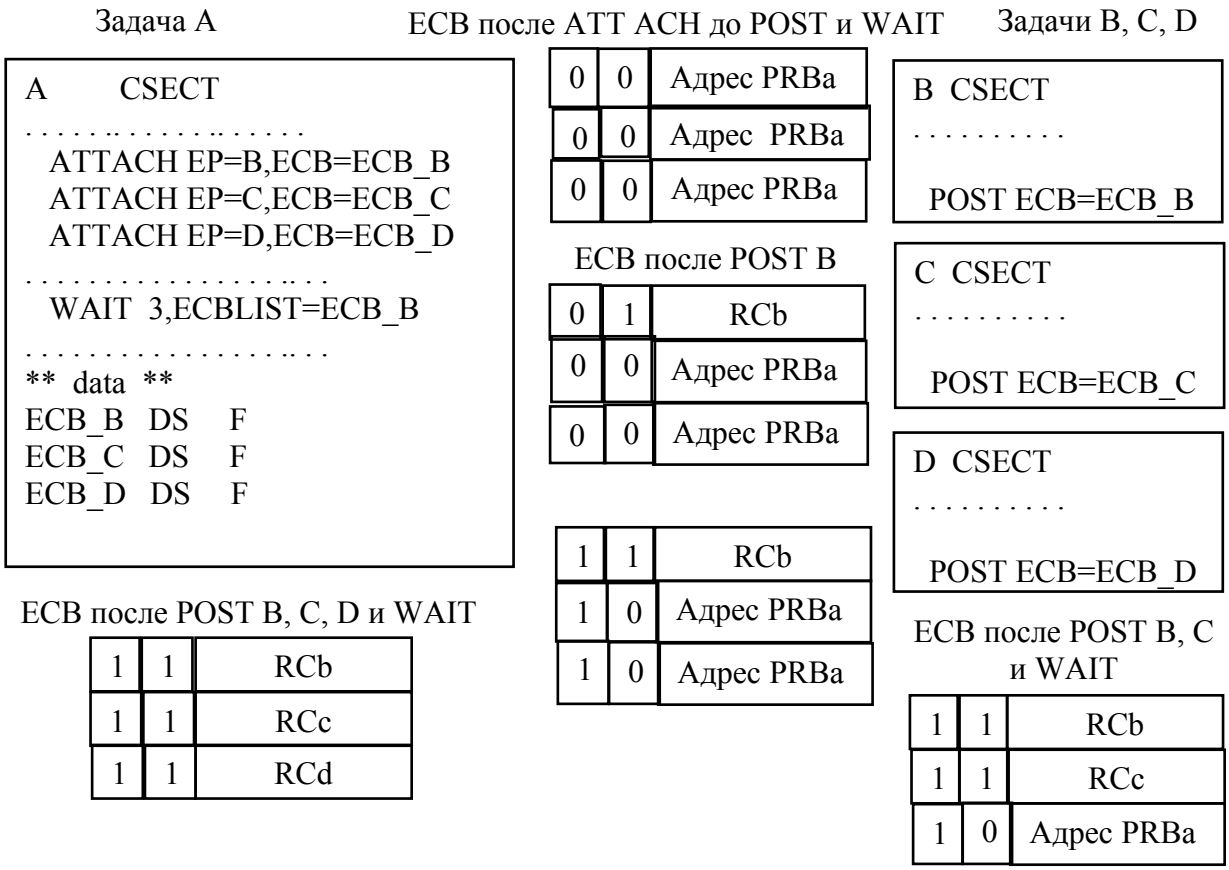


Рис. 77. Пример работы POST и WAIT

Пусть задача A создает 3 подзадачи B, C, D используя макрокоманду ATTACH. В ECB блоках флаги W и P устанавливаются в 0, а поле данных содержит адрес блока запросов программы A.

Также задача B заканчивается раньше чем в задаче A выполнится макрокоманда WAIT. В этом случае POST только устанавливает в1 флаг в ECB блоке.

После завершения задачи B выполняется функция WAIT в программе A которая имеет в качестве первого позиционного параметра число ожидаемых событий. В примере это значение 3. Функция вычисляет реальное число ожидаемых событий вычитая из значения параметра число завершившихся к этому времени событий. В примере это 3-1=2. Это значение записывается в счетчик ожидаемых событий в блоке запросов программы A. Флаг W во всех блоках управления событиями устанавливается в 1 изадача переводится в состояние Ожидания, если реальное число ожидаемых событий не 0.

Далее заканчиваются задачи C и D, в которых выполняется функция POST. В этом случае не только устанавливается в 1 флаг P, но и вычитает из счетчика ожидаемых событий 1, поскольку флаг W=1. Если счетчик получил значение 0, то все события завершены и задача A переводится в состояние Готовность.

8.4.3.5 Синхронизация с помощью ENQ/DEQ в z/OS MVS

Этот механизм предназначен для синхронизации процессов при разделении повторно используемых ресурсов. Выдача макрокоманды ENQ соответствует запросу ресурса. Участок кода от макрокоманды ENQ до макрокоманды DEQ соответствует владению ресурсом, а макрокоманда DEQ соответствует освобождению ресурса.

Ресурс задается двумя именами: Q-именем и R-именем. Q-имя определяет группу однородных ресурсов. R-имя определяет ресурс в группе. Для управления ресурсами используется следующая структура данных (рис. 78).

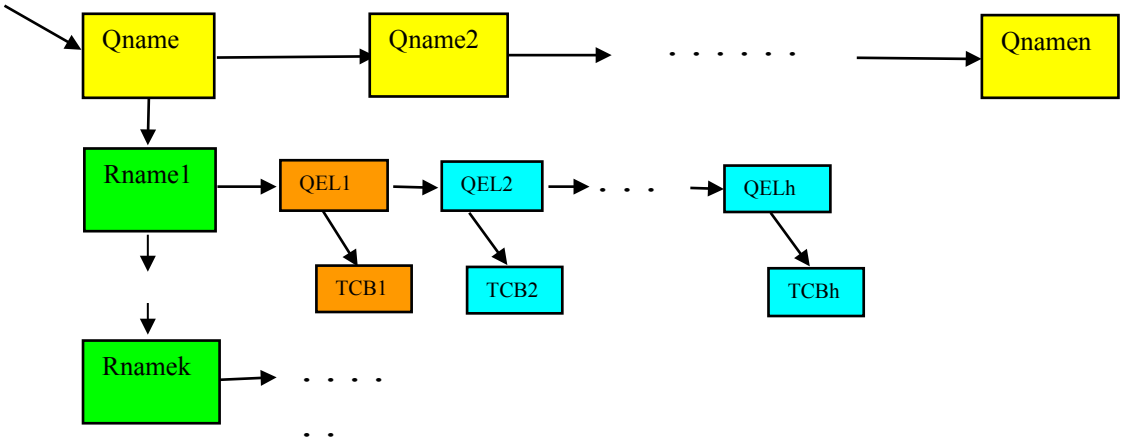


Рис. 78. Структура данных для управления повторно используемыми ресурсами

Эта структура данных динамически строится в области системных очередей SQA (system queue area). Указатель из области ядра указывает на список Q имен. От каждого Q имени строится список R имен, а от каждого R имени строится список запросов к ресурсу. Каждый элемент запроса QEL имеет ссылку на блок управления задачи TCB.

Функция ENQ с параметрами Q-именем и R-именем отыскивает в списке Q имен соответствующее имя. Если оно найдено, то в списке R имен отыскивается R имя. Если ресурс занят, то есть очередь запросов не пуста, то формируется элемент запроса, который ставится в очередь, и задача переводится в состояние ожидания.

Если требуемое Q имя отсутствует, то строятся элемент списка Q имен и элемент списка R-имен. Если Q имя существует, а R-имя отсутствует, то строится недостающее R-имя.

Функция DEQ с параметрами Q-именем и R-именем удаляет из очереди запросов первый запрос, принадлежащий задаче, владеющей ресурсом, и переводит в состояние Готовность задачу для следующего элемента запроса. Это позволяет задаче исполнять код после макрокоманды ENQ, то есть владеть ресурсом.

Разумеется, для практического использования этих макрокоманд следует обратиться к документам фирмы IBM, которые можно скачать свободно с сайта.

8.4.3.6 Синхронизация с помощью сообщений

Этот механизм наиболее развит в QNX. Эта ОС реального времени, сетевая для персональных компьютеров. Эта система может загружаться с любого узла сети и все узлы равноправны.

Сообщение это ресурс, который порождается одним процессом, и потребляется другими процессами, после чего он прекращает свое существование. Сообщение представляет собой последовательность байт и задается в программе как char MSG[256];. В начале сообщения строится блок управления, в который помещается управляющая информация такая как идентификаторы процессов посылающих и принимающих сообщение.

Следующие основные функции поддерживают обмен сообщениями между процессами:

Send() - посылает сообщение другому процессу.

Receive() - принимает посланное сообщение.

Reply() - посылает ответное сообщение.

Эти функции могут быть использованы как локально, то есть для связи между процессами на одном компьютере, так и в пределах сети, то есть для связи между процессами на разных узлах.

Работа этих функций происходит следующим образом. Пусть процесс А посылает сообщение процессу В используя Send(). При этом процесс А переходит в состояние Ожидания. В системе существует несколько вспомогательных состояний ожидания. В этом случае процесс А будет Send-блокирован.

Предположим, что процесс В выдает функцию Receive() после того как процесс А выдал Send(). В этом случае процесс В получает сообщение и продолжает работу. Процесс А переводится в состояние Reply-блокирован. Если процесс В выдает функцию Receive() до того как процесс А выдал Send(), то процесс В переходит в состояние Receive-блокирован до тех пор, пока не прийдет сообщение.

Процесс В производит обработку и в подходящий момент выдает Reply() с ответным сообщением процессу А. Процесс А переходит в состояние Готовность.

Синхронизация с помощью сообщений осуществляется за счет того, что процесс, пославший сообщение будет находится в ожидании, пока не получит ответное сообщение.

Формат функций следующий:

Send(pid, smsg, rmsg, smsg_len, rmsg_len);

pid - идентификатор процесса, которому передается сообщение .

smsg - буфер посылаемого сообщения.

rmsg - буфер ответного сообщения.

smsg_len - длина посылаемого сообщения.

pids = Receive(pidr, msg, msg_len);

pids - идентификатор процесса пославшего сообщения.

pidr - идентификатор процесса, от которого ожидается сообщение. Если 0, то от любого процесса.

msg - буфер для принимаемого сообщения.

msg_len - длина принимаемого сообщения.

Reply(pid, reply, reply_len);

pid - идентификатор процесса, которому передается сообщение .

reply - буфер посылаемого сообщения.

reply_len - длина посылаемого сообщения.

Как видно из форматов функций для их использования необходимо знать идентификатор процесса, которому посылается сообщение или от которого принимается сообщение. Если это родственные процессы, то есть являющиеся предками или потомками друг друга, то идентификаторы можно узнать используя функцию, обходящую дерево процессов. Если процессы не родственные, то дерева процессов не существует. В этом случае необходимо стартовать системное приложение, которое ведет базу данных с именами и идентификаторами процессов. Каждый процесс должен зарегистрироваться в этой базе под некоторым именем и приложение создает запись, в которой имени сопоставляется идентификатор процесса. Если процесс хочет получить идентификатор процесса, то в запросе указывается имя и возвращается идентификатор, соответствующий этому имени.

8.5 Тупиковые ситуации в ОС

При взаимодействии процессов и разделении ресурсов между процессами может возникнуть ситуация, которая называется тупиковой ситуацией или клинчем. Впервые эта ситуация была обнаружена в OS360, когда инициаторы некоторых классов перешли в состояние Ожидания и не обслуживали задания этих классов. Очереди заданий росли и это привело к обнаружению этих ситуаций и их исследованию.

8.5.1 Понятие тупиковой ситуации

В мультипрограммной системе процесс находится в состоянии тупика если он находится в состоянии Ожидания и никогда не может из него выйти. По существу процесс ждет событие, которое никогда не наступит.

Ряд примеров иллюстрирует различные ситуации, возникающие на различных типах ресурсов.

Пример 1. Есть два параллельно выполняющихся процесса, которые запрашивают два повторно используемых ресурса единичной ёмкости. Порядок запросов показан на рис. 79.

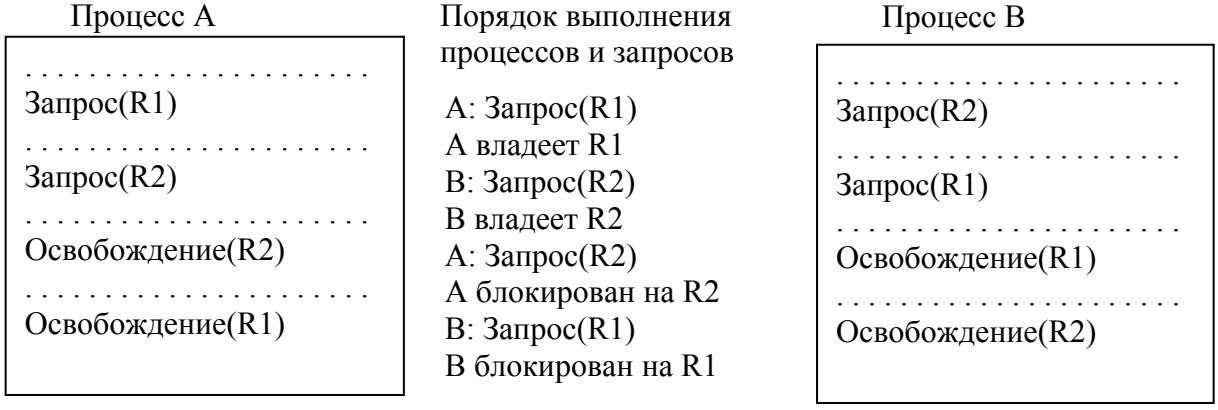


Рис. 79. Пример тупиковой ситуации на ресурсах единичной ёмкости

Как видно из рисунка при определенном порядке выполнения процессов и запросов возникает ситуация, когда два процесса блокировались на совместно используемых ресурсах. Причем ни один из этих процессов никогда не сможет продолжить работу, так как ожидает ресурс занятый другим процессом, но другой процесс не освободит занятый ресурс поскольку находится в состоянии Ожидания. Это типичный пример тупиковой ситуации, называемый "Смертельные объятия".

Пример 2 связан с использованием одного ресурса с n единицами. Пусть единицы этого ресурса запрашивают m процессов, где $m \geq n$. Процессы устроены следующим образом. Каждый процесс запрашивает одну единицу ресурса, затем через некоторое время ещё раз запрашивает единицу ресурса. В этом случае может возникнуть следующая ситуация.

Первый процесс запрашивает единицу ресурса и получает её. Затем другие процессы последовательно запрашивают единицы ресурса и получают их при $m=n$. Если процессов больше, чем единиц ресурса, то часть процессов блокируется. Затем первый процесс опять запрашивает единицу ресурса и блокируется, поскольку все единицы ресурса заняты. Затем другие процессы последовательно запрашивают единицу ресурса и блокируются. Эти процессы никогда не смогут продолжить выполнение, поскольку ни одна единица ресурса не будет освобождена.

Этот пример объясняет почему в универсальных ОС перед загрузкой процесса выделяется определенный объем ОП и далее в процессе работы

процесс не может увеличить этот объем. Память выделяется из области процесса, в котором находится куча и стек.

Пример 3 показывает, что тупиковая ситуация может возникнуть с потребляемыми ресурсами. Пусть существуют три процесса. В каждом процессе сначала принимается сообщение, а затем посылается сообщение.

P1: begin .....Принять(S3);....Послать(S1); end

P2: begin .....Принять(S1);....Послать(S2); end

P3: begin .....Принять(S2);....Послать(S3); end

Предположим, что сначала процесс P1 принимает S3 и при этом блокируется, поскольку S3 еще не послано. Затем процесс P2 принимает S1 и также блокируется поскольку процесс P1 не послал это сообщение. И наконец, процесс P3 блокируется на приеме S2 поскольку процесс P2 его не послал. Также он не пошлет S3 и процесс P1 не будет никогда активен. То же можно сказать и о других процессах. Однако, если сначала посылать сообщение, а затем принимать, то тупика не будет.

Из приведенных примеров видно, что образование тупиковой ситуации зависит от порядка запросов в различных процессах. Поэтому обнаружение возможности возникновения тупика сводится к построению различных комбинаций последовательностей запросов различных процессов во времени. Для различных таких последовательностей запросов можно воспользоваться теорией Холта для анализа тупиков.

8.5.2. Модель Холта для анализа тупиков

Модель Холта позволяет определить для данной ситуации, то есть когда множество процессов совместно использующее ряд ресурсов уже распределило эти ресурсы между собой. В этом случае строится граф определенного вида и в результате преобразования графа можно определить попадут ли какие-либо процессы при дальнейшей работе в состояние тупика.

При разработке мудьти задачного приложения интересует вопрос о возможности тупика при работе такого приложения. В этом случае чтобы воспользоваться моделью Холта придется строить различные возможные ситуации распределения ресурсов между процессаи и для каждой применять теорию Холта.

8.5.2.1 Определение модели

Модель Холта для анализа тупиков представляет собой двудольный граф (биграф), то есть граф, множество вершин которого можно разбить на два подмножества таким образом, что каждое ребро графа соединяет какую-то вершину из одного подмножества с какой-то вершиной другого подмно-

жества и не существует ребра, соединяющего две вершины из одного и того же подмножества.

Одно подмножество соответствует процессам, а другое подмножество ресурсам. Вершины-процессы обозначаются кружками, а вершины-ресурсы обозначаются прямоугольниками с точками внутри. Точки обозначают единицы ресурса. Если ресурс единичной емкости, то точка может отсутствовать. Запрос процессом ресурса обозначается ребром, направленным от вершины-процесса к вершине-ресурсу или к единице ресурса. Владение процессом ресурса обозначается ребром, направленным к вершине-процессу от вершины-ресурса или от единицы ресурса. Эти понятия иллюстрирует рис. 80.

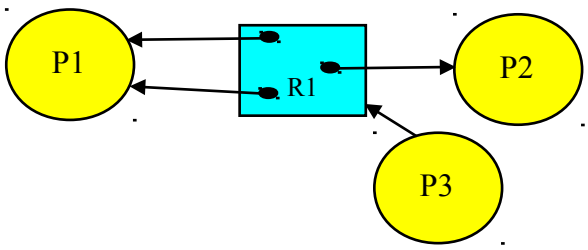


Рис. 80. Модель процессов и ресурсов

Процесс P1 владеет двумя единицами ресурса R. Процесс P2 владеет единицей ресурса R. Оба процесса активны и рано или поздно закончатся. Процесс P3 запросил ресурс R, но все единицы ресурса заняты и процесс блокировался..

Если у вершины-процесса нет ребер, то процесс не владеет и не запросил ресурса.

Модели Холта для примеров 1 и 2 показаны на рис.81.

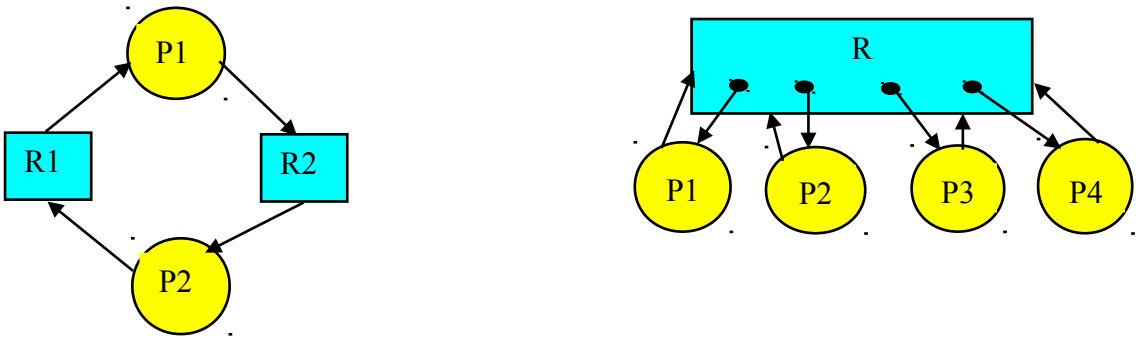


Рис.81. Модели Холта для примеров 1 и 2

8.5.2.2 Процедура сокращения графа

Процедура сокращения графа позволяет выявить наличие тупика для определенной ситуации, которая возникла в определенный момент. Для этой ситуации строится начальный граф, к которому и применяется процедура сокращения. Если в результате получается несвязанный граф, не имеющий ребер, то тупика нет и все процессы благополучно закончатся. Если на каком-

то шаге получится несокращаемый граф, то дальнейшее развитие процессов приведет к тупику.

Процедура сокращения графа состоит в следующем.

1) Найти активный процесс в графе, то есть вершину-процесс, в которую входят ребра и нет исходящих ребер. Если таких вершин нет, то все процессы блокированы и это тупик.

2) Для найденных вершин активных процессов удаляются все ребра. В результате происходит освобождение ресурсов. Если в графе после удаления ребер не осталось, то процедура заканчивается и тупика нет.

3) Если к освободившемся ресурсам существуют запросы других процессов, то ребра меняют свое направление. Перейти к п. 1).

Пример применения процедуры сокращения показан на рис. 82.

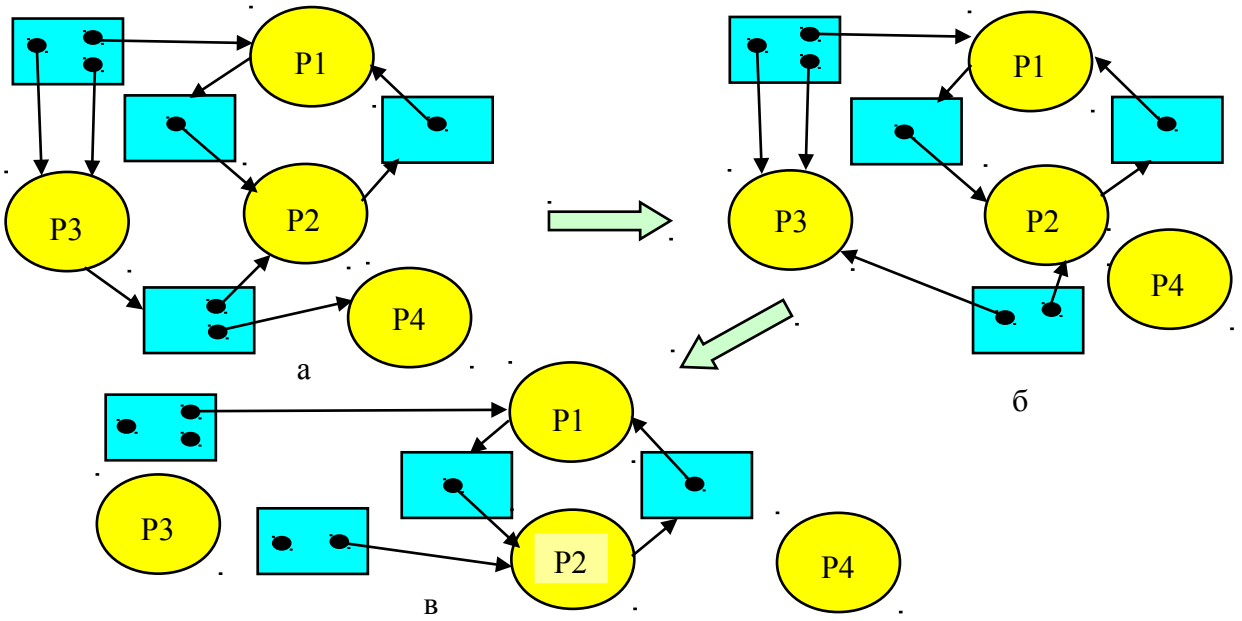


Рис. 82. Пример работы процедуры сокращения графа

В начальном графе (рис. 82а) процесс P4 является активным. Поэтому в определенный момент времени закончится и освободит единицу ресурса. Процесс P3 (рис. 82б) получает запрошенную единицу ресурса и становится активным. В результате ребра к P3 уничтожаются и получается граф рис. 82в. Этот граф не может быть далее сокращен. Поскольку в результирующем графе есть связанные вершины, то эта начальная ситуация приводит к тупику.

8.5.3 Распознавание, восстановление и предотвращение тупиков

ОС можно строить в соответствии с двумя принципами: допускать наличие тупиков или предотвращать их, разрабатывая безопасную систему. Если в системе допускаются тупиковые ситуации, то достигается большее быстродействие работы системы, но необходимы функции распознавания тупика и восстановления работоспособности после обнаружения тупика. Такой подход используется в управляющих системах, системах реального времени.

Универсальные системы разрабатываются как безопасные системы и в них используются специальные алгоритмы, предотвращающие тупиковые ситуации. Однако, современные прикладные программы строятся как мультипрограммные приложения, которые состоят из взаимодействующих процессов и в этом случае вся ответственность лежит на разработчиках. Поэтому программисты должны владеть этой проблемой.

Распознавание тупика связано с анализом процессов, совместно использующих ресурсы. Нужно найти ситуации, когда такие процессы блокируются при запросе ресурсов. Затем можно построить граф модели Холта и использовать процедуру сокращения графа для обнаружения существования тупика.

Восстановление работоспособности процессов находящихся в тупике связано с освобождением ресурсов, занятых процессами. Это можно сделать уничтожив определенный процесс или передав ресурс от одного процесса другому процессу. Такие действия возможны в системах с фиксированным набором задач, когда известны все возможные ситуации.

Предотвращение тупика используется в универсальных системах для построения безопасных систем. В этом случае при распределении ресурсов используются методы предотвращающие тупики.

Метод предварительного распределения ресурсов предполагает до запуска процесса резервирования всех необходимых ему ресурсов. Это возможно сделать, если, во-первых, существует описание всех ресурсов для процесса. Во-вторых, необходимо иметь системное средство распределяющее эти ресурсы. И, наконец, невозможно для всех процессов как системных, так и прикладных применить предварительное распределение, поскольку в этом случае потребуется некоторый верховный распределитель и мультипрограммный режим будет весьма ограничен.

Тем не менее такой метод используется в z/OS MVS для задач шагов задания. Для каждого шага задания в операторе EXEC указывается программа, которая будет выполняться в рамках задачи и в последующих операторах DD описываются используемые наборы данных, устройства. Эту информацию использует системная задача инициатор INIT, которая размещает ресурсы и потом запускает задачу шага задания. Но инициаторов может быть 256 и все они конкурируют за ресурсы. Поэтому тупиковые ситуации могут возникать среди этих системных задач. Кстати, впервые тупиковая ситуация была обнаружена среди инициаторов в первых системах OS360. Поэтому в инициаторах используется контроль за распределением ресурсов.

Одним из методов контроля за распределением ресурсов является стандартная последовательность распределения. В соответствии с этим методом все ресурсы разбиваются на классы $К_1$, $К_2$,...,$К_n$. То есть каждый ресурс принадлежит определенному классу. Ресурсы выделяются в порядке возрастания классов, то есть сначала выделяются ресурсы, принадлежащие младшему классу, затем следующему по возрастанию. Если ресурс не может быть выделен для какого-то класса, то ресурсы для последующих классов не выделяются. Такой метод позволяет избежать тупиковых ситуаций.

Таким образом применение методов предотвращения тупиков снижает быстродействие системы. Задачи вынуждены ждать освобождения ресурсов. Но это компенсируется безопасностью системы.

Тема 9. УПРАВЛЕНИЕ УСТРОЙСТВАМИ В ОС

Рассматриваются внешние устройства компьютера. Многие устройства различной конструкции и различных производителей имеют с точки зрения управления схожие характеристики и методы управления ими имеют много общего.

Центральный процессор и основная память также являются важнейшими устройствами компьютера, но требуют собственных методов управления, которые были рассмотрены в предыдущих темах.

Рассматриваемые в этой теме устройства называются внешними или периферийными.

9.1 Функции системы управления устройствами

Система управления устройствами выполняет следующие функции.

1) Отслеживание состояния устройств.

Информация о каждом устройстве хранится в блоке управления устройством UCB (unit control block). В блоке хранится адрес устройства, тип устройства и другая информация, характеризующая устройство. Также там хранится переменная информация о состояниях устройства. Состояния устройства могут быть ONline/OFFline, READY/NOTREADY. Эти состояния используются при репликации информации, когда запись на запоминающее устройство копируется на другое устройство. В этом случае второе устройство должно быть защищено от записи со стороны и переводится в состояние OFFline. Часто в системах в случае сбоев устройство переводится в состояние NOTREADY.

Блоки управления устройствами образуют список и при поиске устройства система ввода/вывода сканирует список. Устройства могут добавляться или исключатся. В этом случае список блоков управления редактируется.

2) Планирования использования устройств.

Планирование определяет какой процесс в какой момент времени и на какой срок получит доступ к устройству. Существуют следующие способы предоставления устройства процессу.

Монопольное использование устройства процессом. В этом случае устройство предоставляется только одному процессу на время его жизни или на меньшее время, но процесс повторно не запрашивает доступ к этому устройству. Характерным устройством, которое используется монопольно, является накопитель на магнитной ленте.

Разделение устройства между процессами.

Устройства могут использоваться совместно несколькими процессами или разделять устройство на последовательной основе. В качестве примера совместно используемого устройства можно привести PAV (parallel access volume) том с параллельным доступом. Устройства разделяемые на последовательной основе это диски.

Виртуальные устройства.

Использование виртуальных устройств применяется для таких реальных устройств, которые невозможно или неэффективно использовать для каждого пользователя. Такое устройство следует иметь одно, но использовать могут многие пользователи. Пользователь полагает, что работает с реальным устройством, но на самом деле используется другое физическое устройство. Так если принтер неэффективно ставить к каждому компьютеру, то информация собирается в специальный файл на диске и после окончания работы выводится из файла на принтер.

3) Управление устройствами.

Выделение устройства процессу и освобождение устройства относятся к функциям управления. Обычно эти функции выполняет система ввода/вывода. Выделение связано с поиском устройства на котором размещен файл или набор данных. Затем адрес найденного устройства помещается в таблицы используемые системой ввода/вывода. В некоторых системах указывается символическое имя устройства и по списку UCB отыскивается адрес. Если создается новый файл или набор данных, то либо также по символическому имени отыскивается адрес, либо выбирается устройство из числа предназначенных для размещения данных.

Освобождение устройства состоит в уничтожении соответствующих таблиц.

9.2 Характеристики внешних устройств

Способы управления зависят от характеристик периферийных устройств. Существует большое разнообразие таких устройств. В первую очередь их разделяют на две группы: байт-ориентированные и блок-ориентированные.

Байт-ориентированные устройства не запоминают информацию. Это клавиатура, экран, принтеры. В некоторых системах экран и клавиатура объединяются в виртуальный терминал, но обмен с ним также ведется байтами. Информация посылается на эти устройства или считывается с устройства и повторно недоступна. Эти устройства также называют устройствами с нефайловой структурой или устройствами ввода/вывода.

Блок-ориентированные устройства являются запоминающими и информация пишется и считывается блоками. Это диски, накопители на магнитной ленте, магнитный барабан. Существует флэш память, которая является массовой памятью с адресами, но работа с ней происходит по интерфейсу как с диском.

Важнейшей характеристикой запоминающихся устройств является время доступа к информации, вернее разброс этого времени, который характеризует способ хранения информации на носителе и способ доступа. По времени доступа устройства различаются на следующие:

1) Устройства с постоянным временем доступа, в которых нет разброса времени доступа. Это устройства массовой памяти, где время тратиться на

вычисление адреса. Это флэш память, к которой обращаются как к диску, но реально читается по адресу памяти.

2) Устройства с небольшим разбросом времени доступа. Это дисковые накопители. Устройство диска показано на рис. 83а. Дисковый накопитель представляет собой 15 дисков, вращающихся на шпинделе. На каждом диске на одинаковом расстоянии от центра располагаются дорожки, на которые записывается информация. Каждая порция информации представляет собой физическую запись. Считывающие головки расположены на решетке на одинаковом расстоянии от центра. Таким образом каждая головка располагается

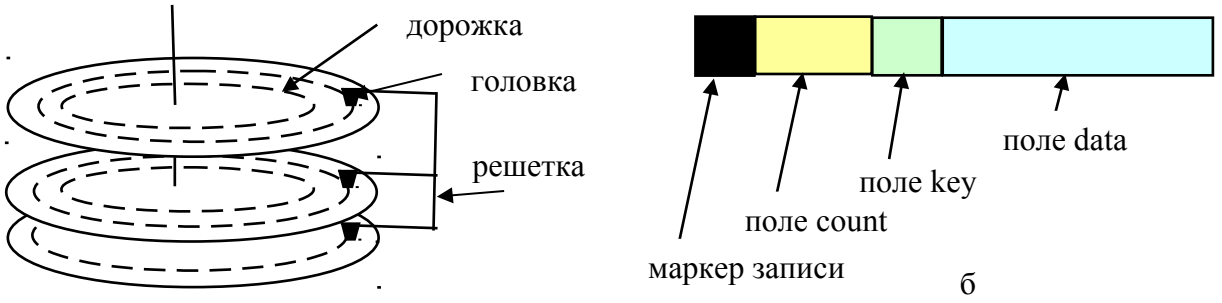


Рис. 83. Структура дискового накопителя и физической записи

над дорожкой с одинаковым номером. Совокупность дорожек равноудаленных от центра на всех 15 дисках называется цилиндром.

Таким образом адрес записи имеет следующий формат CCCCHHR, где CCCC адрес цилиндра, HH адрес головки (0...E), R номер записи на дорожке. Формат физической записи показан на рис. 83б. Маркер записи позволяет определить начало записи и номер. Поле count содержит управляющую информацию, размер поля данных и флаг наличия поля ключа. Поле key может отсутствовать или содержит ключ записи. Поле data содержит данные. Дисковые накопители различаются размерами и это выражается в различном числе цилиндров.

Разброс времени доступа к информации связан с различными адресами цилиндров в запросе и текущим положением головок. Решетка должна перемещать головки от центра к краям и обратно для того, чтобы найти нужный цилиндр. Кроме того нужная запись при вращении диска должна стать под головку поэтому тратиться время на ожидание записи. Выбор дорожки на цилиндре связан с определением головки, которая будет читать или писать физическую запись и производится электронным способом. Это время постоянно и очень мало по сравнению с механическим перемещением решетки и ожиданием записи.

Эти дисковые накопители называются CKD (count-key-data) устройствами по формату физической записи. Они используются на мэйнфреймах. В Open System используются FBA (fix block area) устройства. Они используют физические записи фиксированной длины 512 байт, которые называются блоками. Обращение к такому устройству происходит по номеру блока, а этот номер внутри устройства преобразуется к виду с адресами цилиндров, головок (дорожек) и адреса записи на дорожке.

3) Устройства с большим разбросом времени доступа. К таким устройствам относится накопитель на магнитной ленте. Формат записи на ленту показан на рис. 84.

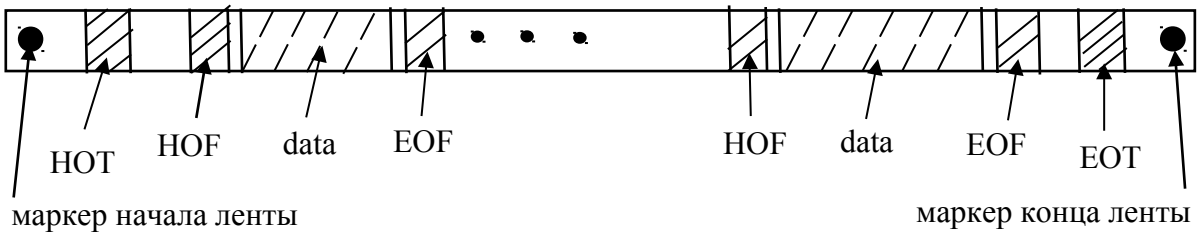


Рис. 84. Формат записи на ленту

Маркеры начала и конца ленты служат для остановки движения ленты. В начале ленты создается заголовок HOT (head of tape), а в конце EOT (end of tape). В них хранится дата создания, модификации, имя владельца и другая управляющая информация. Между заголовком и концом ленты располагаются файлы. Каждый файл имеет заголовок HOF (head of file) и конец EOF (end of file). В них записывается имя файла, дата создания и другая информация. Между заголовком и концом файла записывается содержимое файла.

Рассмотренный формат представления информации на ленте наиболее популярен, поскольку при порче участка ленты теряется не вся информация, а только ее часть. Существовали и ленты другого типа, когда пространство для записи размечалось адресами. В этом случае можно читать и писать информацию по адресам, что существенно быстрее, но порча участка приводила к потере всей информации, поскольку необходимо было переразмечать ленту.

Разброс времени при такой организации размещения информации существенно больше, поскольку необходимо последовательно прочитать заголовки всех предыдущих файлов, прежде чем отыскать необходимый.

Таким образом массовая память и диски являются устройствами прямого доступа, а магнитная лента требует последовательного доступа.

9.3 Организация работы с внешними устройствами

Работа с внешними устройствами зависит от организации системы ввода/вывода компьютера. Различают прямой ввод/вывод, когда процессор имеет непосредственный доступ к контроллеру устройства и косвенный ввод/вывод, когда устройством управляет периферийный процессор. В этом случае ЦП имеет команды для управления периферийным процессором.

9.3.1 Прямой ввод/вывод

Прямой ввод/вывод используется в КС малой и средней мощности. Он базируется на шине, которая представляет совокупность адресных линий, линий данных и управляющих линий. К шине присоединяются контроллеры устройств CU, ОП и ЦП (рис.85).

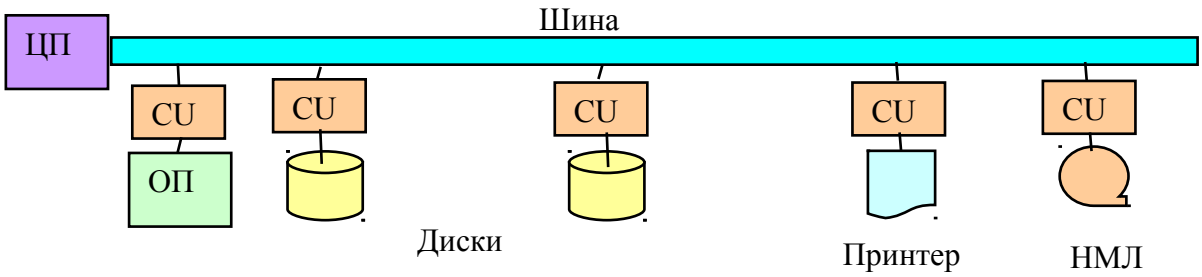


Рис. 85. Шинная организация системы ввода/вывода

В КС используется несколько типов шин, поскольку внешние устройства имеют различные характеристики и обладают различным быстродействием. Так накопители на магнитных дисках используют более быстродействующую шину, а принтеры или терминал используют более медленную шину. Обмен информацией по шине осуществляется следующим образом.

Для каждого типа устройств имеются программы драйверы, которые и осуществляют обмен информацией между буфером ОП и контроллером устройства. Драйверу в числе других параметров передаются адрес устройства и адрес буфера ОП. Драйвер захватывает шину и устанавливает связь с контроллером требуемого устройства. Затем он посылает команды контроллеру устройства, необходимые для выполнения заданной операции. После посылки очередной команды драйвер ждет сигнала от контроллера об окончании выполнения команды. Это ожидание может быть реализовано двумя способами. Драйвер может периодически опрашивать выход контроллера об окончании команды. В этом случае ЦП занят выполнением команд драйвера и недоступен другим программам. Такой способ использовался в маломощных КС. С повышением производительности ЦП стал использоваться другой способ. Устройству выделяется вектор прерывания, в который записывается адрес драйвера. После выдачи команды драйвер переходит в состояние Ожидания. Контроллер устройства после окончания выполнения команды возбуждает прерывание с заданным вектором. Управление передается драйверу и он посылает следующую команду.

Таким образом определенное время процессора занято на выполнение команд драйвера, то есть ЦП непосредственно занят вводом/выводом. Это существенно влияет на производительность системы.

9.3.2 Косвенный ввод/вывод

Организация ввода/вывода с периферийными процессорами освобождает ЦП на все время обмена информацией с внешним устройством. ЦП только запускает периферийный процессор, а затем может быть использован другими программами. Когда ввод/вывод заканчивается, периферийный процессор генерирует прерывание.

Впервые такая организация была разработана в системе IBM360. Далее во всех последующих системах этот подход развивался и в настоящее время используется на мэйнфреймах z/Series.

Периферийные процессоры в этих системах называются каналами. Они связывают логический раздел LPAR (logical partition) и внешние устройства (рис. 86). LPAR является виртуальной машиной, на которую ставится операционная система. Такой способ позволяет управлять обширными ресурсами мэйнфрейма. Логический раздел – поднабор процессорного оборудования, выделенный для поддержки операционной системы.

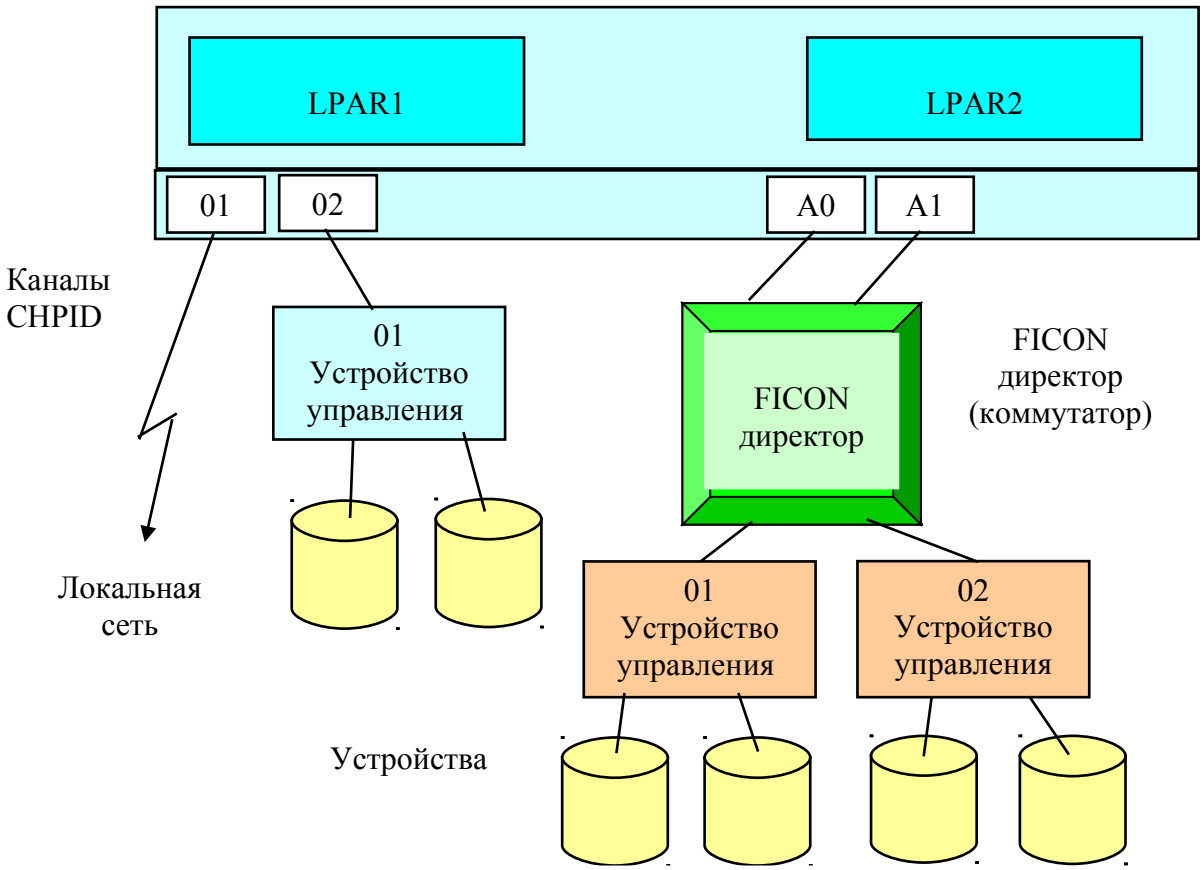


Рис. 86. Система ввода/вывода мэйнфреймов

В настоящее время используются FICON-каналы, которые являются последовательным интерфейсом и используют оптоволоконные соединения. Таких каналов может быть до 256. FICON-каналы подключаются только к одному устройству или одному порту коммутатора. В большинстве современных мэйнфреймах используются коммутаторы между каналами и устройствами управления. Коммутаторы могут подключаться к нескольким разделам, совместно используя устройства управления и некоторые или все их устройства ввода/вывода во всех системах. Таким образом образуется путь к устройству ввода/вывода через канал, коммутатор и устройство управления (контроллер устройства).

Канал характеризуется CHPID –идентификатор канального пути (channel path identifier). Несколько разделов иногда могут совместно использовать CHPID. Это зависит от типа устройств управления, используемых через CHPID. В целом CHPID, используемые для дисков, допускают совместное использование. Физический адрес устройства формируется из адреса CHPID, адреса порта коммутатора, адреса устройства управления и адреса самого устройства. Однако использовать физический адрес в ОС неудобно и поэто-

му устройства нумеруются от 0000 до FFFF. Эти номера назначаются при настройке системы и преобразуются в физические с помощью таблицы IOCDS (I/O Control Data Set), Эта таблица загружается в HSA (Hardware Save Area).

Каналы, как периферийные процессоры, используют свою собственную систему команд, а не систему команд ЦП как драйверы при прямом вводе/выводе. Команды канала называются CCW (channel command word). Для осуществления операции обмена строится канальная программа состоящая из CCW. Канальную программу может построить программист, но это очень сложно сделать, поскольку руководство по командам является закрытой информацией IBM. В основном используется система ввода/вывода, программы которой настраивают шаблоны канальных программ на конкретную операцию обмена.

9.4 Реализация функций управления внешними устройствами

Реализация функций управления зависит от характера использования внешних устройств.

9.4.1 Монопольно используемые устройства

При монопольном использовании устройства функции реализуются весьма просто. Учет состояния устройства "свободно-занято" может отмечаться флагом в блоке управления устройством. Планирование занятия и освобождения устройства осуществляется по запросу. При запросе устройства отыскивается путь к устройству, проверяется свободно ли устройство и строятся таблицы, которые используются системой ввода/вывода. При освобождении устройства таблицы уничтожаются и помечается, что устройство свободно.

9.4.2 Повторно используемые устройства

В случае повторно используемых устройств, разделяемых последовательно функция учета связана с поиском таблиц, описывающих путь к устройству, созданием элементов запроса и постановкой в очередь этих элементов. Так в случае косвенного ввода/вывода существует таблица логических каналов, по которой находят канальный путь к устройству.

По найденному канальному пути находится блок управления каналом, который содержит имя канала, состояние канала, список устройств управления, доступных по этому каналу, список задач, ожидающих канал и состоящий из запросов.

Элементы списка устройств управления указывают на блоки управления устройством управления. Такой блок содержит информацию о состоянии устройства управления, адрес устройства управления, список устройств, список каналов, к которым подключено устройство управления, список задач, ожидающих устройство управления.

Элемент списка устройств указывает на блок управления устройством, который содержит адрес устройства, состояние устройства, список устройств

управления, к которым подключено устройство, очередь запросов ожидающих задач.

Планирование связано с управлением очередью запросов. Элемент очереди запросов к диску содержит адрес физической записи в виде CCCCHHR, где CCCC адрес цилиндра, HH номер головки, R адрес записи. Поэтому очередь запросов для устройств большой емкости оптимизируют по времени доступа к информации. Различают два способа оптимизации: по времени поиска цилиндра и времени ожидания записи.

Оптимизация по времени поиска цилиндра состоит в упорядочении очереди запросов по адресу цилиндра CCCC и текущему положению головки относительно цилиндра. Следующий пример иллюстрирует эту проблему. Пусть головка расположена над 5 цилиндром. А в очереди запросов расположены адреса в следующем порядке: 10 цилиндр, 7 цилиндр, 20 цилиндр, 15 цилиндр. Если исполнять запросы последовательно (алгоритм FIFO), то головка продвинется через 5 цилиндров к 10, затем вернется на 3 цилиндра к 7, затем продвинется на13 цилиндров к 20 и наконец вернется через 5 цилиндров к 15. Таким образом головка в общей сложности пройдет 26 цилиндров, чтобы прочитать все записи. Если упорядочить запросы по возрастанию, учитывая что головка расположена на самом меньшем цилиндре, то есть в порядке 7 цилиндр, 10, 15 и 20 цилиндры, то головка в общей сложности переместится через 2, 3, 5 и 5 цилиндров, чтобы прочитать все записи. В общей сложности через 15 цилиндров, а не 26.

Известны следующие алгоритмы оптимизации по времени поиска цилиндра.

1) SSTF (shortest seek time first) с наименьшим временем поиска первым. По этому алгоритму выбирается запрос требующий минимального перемещения головки. Недостатком этого алгоритма является дискриминация крайних цилиндров. Поэтому запросы обслуживаются неравномерно, запросы к крайним цилиндрам обслуживаются гораздо дольше. Этот алгоритм используется в системах с умеренной нагрузкой.

2) SCAN алгоритм сканирования В этом случае головка движется в одном направлении, затем в другом. Для обслуживания выбираются запросы с минимальным временем подвода в привилегированном направлении. Этот алгоритм обеспечивает меньший разброс времени в обслуживании запросов.

3) N-step SCAN алгоритм работает как предыдущий, но обслуживаются только запросы уже пришедшие к началу движения в том или другом направлении. Если запрос пришел во время движения головки, то он на этом проходе не рассматривается. Этот алгоритм обеспечивает еще меньший разброс времени.

4) C-SCAN алгоритм циклического сканирования. В этом алгоритме запросы обслуживаются только при движении головки от наружных к внутренним цилиндрам. Затем головка просто выводится от внутренних к наружным цилиндрам. Этот алгоритм характеризуется самым меньшим разбросом времени обслуживания.

9.4.3 Виртуальные устройства

Виртуальные устройства используются чтобы заместить такие реальные устройства, которые нецелесообразно ставить в большом количестве в КС. Кроме того эти устройства требуют монопольного обслуживания, но используются многими процессами.

Впервые виртуальные устройства были использованы в OS360. Так программа системный ввод была предназначена для ввода заданий с устройства ввода с перфокарт. Но КС становились все более мощными и быстродействующими, а устройство ввода с перфокарт принципиально не могло быть быстродействующим. Поэтому была разработана программа SPOOLer (Simultaneously Peripheral input/Output On Line), которая вводила перфокарты и писала образы перфокарт в виде 80 байтовых записей в системный набор данных. Эта программа монопольно использовала устройство ввода с перфокарт. Программа системный ввод читала задания из набора данных как с устройства ввода с перфокарт. В современных системах z/OS MVS также используется ввод во входной поток JES (job entry system) с терминалов и по сети. Это позволяет получать доступ к мэнфрейму из любой точки земного шара.

Таким же образом устроена и система вывода. Задания различных классов одновременно выводят информацию, но если они будут использовать реальное устройство, то строки разных заданий будут перемешаны. Поэтому необходимо собирать всю информацию по заданию и лишь затем выводить на реальное устройство. Для этого в OS360 вывод заданий происходил в системный набор данных, в котором для каждого задания строилась структура данных. Затем программа системный вывод по команде оператора выводила сгруппированные данные на реальное устройство печати, которым эта программа монопольно владела.

Таким образом при использовании виртуальных устройств существует ПО, которое реализует интерфейс к виртуальным устройствам как к реальным. Эти программы используют структуры данных, с помощью которых ведут учет и планирование виртуальных устройств.

Тема 10. УПРАВЛЕНИЕ ВВОДОМ/ВЫВОДОМ В ОС

Реализация функций ввода/вывода в ОС зависит от реализации ввода/вывода в КС. Поскольку существуют два способа организации в виде прямого и косвенного способов, то и реализация функций ввода/вывода для этих двух способов принципиально различаются.

Ввод/вывод на байт ориентированные устройства весьма специфичен и реализуется в системе как стандартный ввод или стандартный вывод. Программист редко нуждается в изменении этих функций. Как правило система позволяет легко перенаправить стандартный ввод/вывод на другие устройства. А вот организация ввода/вывода на запоминающие устройства требуют существенного участия программиста. Поэтому в этой теме рассматривается ввод/вывод на запоминающие устройства.

Система ввода/вывода ОС получает в качестве параметров адреса буфера в ОП и области на запоминающем устройстве. Функция системы состоит в передаче блока записей между ОП и устройством и проверка правильности передачи. Если передача выполнена неверно, то она может быть повторена несколько раз.

10.1 Реализация функций управления вводом/выводом в z/OS

В пакетной ОС используются КС имеющие косвенный ввод/вывод. Это связано с необходимостью обеспечения эффективного обмена информацией с устройствами. Поэтому система ввода/вывода является весьма сложной. Кроме того программист должен понимать организацию и работу системы поскольку в задании он должен описать устройства и наборы данных, используемые в каждом шаге задания.

10.1.1 Общая схема ввода/вывода в z/OS

На рис. 87 представлена общая схема ввода/вывода, который происходит с ввода задания в систему.

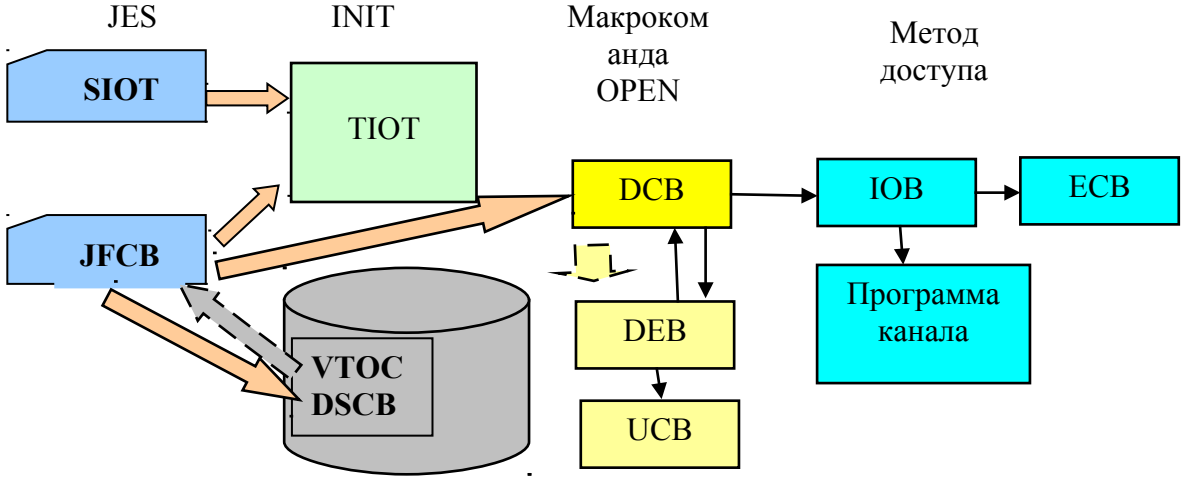


Рис. 87. Общая схема ввода/вывода в z/OS

При вводе задания в систему JES по тексту задания на JCL создает совокупность таблиц. Для всего задания создается JCT (job control table) таблица управления заданием. Для каждого шага задания (оператор EXEC в JCL) создается SCT (step control table) таблица управления заданием. А для каждого описания набора данных создаются две таблицы. Это таблица ввода/вывода шага задания SIOT (step input/output table) и блок управления файлом задания JFCB (job file control block). На рис. 87 рассматривается организация ввода/вывода для одного НД. Для остальных НД шага задания будут строится такие же таблицы и выполнятся аналогичные действия.

При выполнении шага задания инициатор INIT строит таблицу ввода/вывода задачи TIOT (task input/output table), куда помещает данные из JCT, SCT и для каждого набора данных шага задания из таблиц SIOT и JFCB. Также инициатор размещает НД шага задания. Если НД новый, то отводится место требуемого размера на диске (параметр SIZE= в DD операторе) и создается дескриптор НД DSCB (data set control block) в таблице тома VTOC (volume table of content). Если набор данных существует, то информация из DSCB переписывается в SIOT и JFCB. Таким образом до запуска программы собирается вся информация о наборах данных шага задания. После этого инициатор запускает программу шага задания.

Программа для работы с НД выполняет макрокоманду OPEN, которая в качестве параметра принимает адрес DCB. При выполнении макрокоманды управление передается одноименной функции супервизора. Эта функция используя информацию из TIOT, SIOT, JFCB, DSCB и других строит блок экстентов данных DEB (data extent block). В этом блоке записаны адреса участков памяти на диске, в которых расположен НД. Действительно, данные могут быть записаны на один сплошной участок памяти или в разные несмежные участки памяти диска. Поэтому пространство памяти на диске занимаемое НД может состоять из одного или нескольких экстентов. Также отыскивается UCB в списке блоков для устройства с НД.

После открытия НД в программе выполняются макрокоманды чтения или записи, которые осуществляют обмен информацией с устройством. Эти макрокоманды содержат обращения к методам доступа, то есть к системным программам, которые грузятся в адресное пространство задания. В качестве параметра методу доступа передается адрес DCB.

Метод доступа используя собранную информацию строит блок ввода/вывода IOB (input/output block), канальную программу и для синхронизации с программой блок управления событием ECB. Затем происходит обращение к физической системе ввода/вывода путем вызова функции EXCP (execute channel program). Эта функция супервизора запускает канал, который исполняет построенную канальную программу. А программа метода доступа выдает макрокоманду WAIT и задача осуществляющая ввод/вывод переходит в состояние *Ожидания*.

После завершения операции ввода/вывода EXCP выдает макрокоманду POST о завершении события, задача разблокируется и продолжает выполняться.

Таким образом ввод/вывод подготавливается на всем протяжении выполнения задания, начиная с его ввода во входной поток JES и заканчивая супервизором EXCP.

10.1.2 Физическая система ввода/вывода в z/OS

Супервизор EXCP является физической системой ввода/вывода. А совокупность методов доступа называют логической системой. Супервизор выполняет две основные функции:

1) Обрабатывает запросы на ввод/вывод.

2) Обрабатывает прерывания от канала по окончанию операции обмена.

Соответственно функция EXCP имеет два входа: по обращению к супервизору с запросом на ввод/вывод и по прерыванию от канала по окончанию выполнения канальной программы.

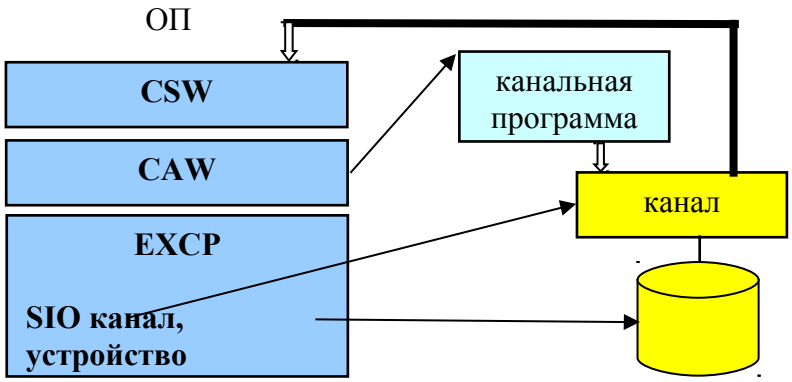


Рис. 88. Организация ввода/вывода

На рис. 88 показана организация ввода/вывода в системе. EXCP супервизор отыскивает путь к устройству через канал и контроллер. Когда путь найден в CAW заносится адрес канальной программы и исполняются команды проверяющие канал и запускающие его работу. Команда SIO (start input/output) инициирует работу указанного канала с заданным устройством.

Канал используя поле CAW находит канальную программу и исполняет ее. Команды CCW канальной программы содержат адреса буфера в ОП и области на устройстве, между которыми осуществляется обмен. После выполнения канальной программы канал записывает состояние окончания обмена в CSW (channel status word) – слово состояния канала. Затем канал возбуждает прерывание, которое обрабатывает EXCP супервизор.

Супервизор EXCP получает управление в результате синхронного (программного) прерывания, получает в качестве параметра адрес IOB и обрабатывает запрос на ввод/вывод следующим образом.

1) Осуществляется проверка IOB.

2) Создается элемент запроса на ввод/вывод.

3) Просматриваются блоки устройств UCB и отыскивается запрошенное устройство. Определяется доступно ли оно.

4) Для получения физического канала к устройству запускается программа проверки канала. Она по таблице логических каналов определяет физический канал.

5) По таблице устройств выбирается программа начала ввода/вывода и ей передается управление.

6) Если операция ввода/вывода не может быть начата, то запрос ставится в очередь. В противном случае запускается операция обмена.

7) После обработки запроса управление передается функции ядра Диспетчер.

Запущенные операции ввода/вывода в какой-то момент заканчиваются. Канал возбуждает прерывание. Эти прерывания асинхронные (аппаратные) и могут возникать в произвольные моменты времени. В результате обработки прерывания управление получает супервизор ввода/вывода. В этом случае он выполняет следующие функции.

1) По списку UCB отыскивается устройство, для которого закончился ввод/вывод.

2) Анализируются биты состояния установленные по окончанию операции обмена.

3) Если операция закончилась успешно, то выдается макрокоманда POST, которая выводит из состояния ожидания программу, запросившую ввод/вывода. Если операция закончилась неудачно, то она может быть повторена несколько раз. В случае неудачного завершения и в этом случае операция также заканчивается и выдается макрокоманда POST.

4) Просматривается очередь запросов на ввод/вывод и запускаются возможные операции.

5) Прерывания ввода/вывода, которые пришли во время обработки прерывания, сохраняются. Поэтому для этих прерываний выполняются действия с пункта 1.

6) Если все прерывания обработаны и нет запросов на ввод/вывод, которые могут быть удовлетворены, то осуществляется переход к Диспетчеру.

Таким образом физическая система ввода/вывода осуществляет физический обмен с устройствами используя физические адреса.

10.1.3. Логическая система ввода/вывода в z/OS

Логическую систему ввода/вывода составляют методы доступа. Это программы, которые грузятся в область памяти задачи за исключением физического метода EXCP, который реализуется функцией ядра.

Метод доступа – сочетание способа доступа к наборам данных с определенным типом организации данных. Существуют следующие типы организации НД: последовательные, библиотечные, с прямым доступом, расширенные (Extended DS) и VSAM (virtual storage access method).

Способы доступа могут быть с очередями и базовые.

Способы доступа с очередями применяется только к НД, в которых адрес следующей записи может быть определен по предыдущей записи. Это последовательные НД или разделы библиотек.

Программист имеет дело с логическими записями. Макрокоманда GET читает в программу логическую запись, PUT выводит логическую запись в

буфер. Программист может определить размер буфера, количество буферов в буферном пуле и тип буферизации.

При чтении НД по макрокоманде OPEN буфера загружаются из НД до команды чтения логической записи GET. Из НД читается физическая запись, которая содержит блок логических записей. По мере освобождения буфера блоки из НД подкачиваются и разблокируются. Такой способ буферизации называется буферизацией с упреждением.

При записи НД буфера заполняются программой по макрокоманде PUT и заполненный буфер выгружается в НД автоматически без участия программиста.

В методах доступа с очередями синхронизация выполняется программой метода доступа. Сразу после макрокоманды EXCP в программе метода доступа выдается WAIT. Когда EXCP супервизор заканчивает операцию обмена, он выдает POST. Также программа метода доступа проверяет ошибки ввода/вывода и возвращает программе код завершения.

После чтения или вывода логической записи макрокомандами GET или PUT управление возвращается программе из программы метода доступа только после помещения логической записи в буфер.

Базисные методы доступа применимы ко всем типам НД. Используя базисные методы доступа программист осуществляет обмен на уровне блоков. Макрокоманды READ и WRITE читают и пишут блоки логических записей. Поэтому блокирование и разблокирование записей, синхронизацию с EXCP супервизором, обработку ошибок ввода/вывода, управление буферами осуществляет программист. Метод доступа составляет канальную программу, строит IOB и обращается к EXCP супервизору. На рис. 89 показано использование методов доступа.

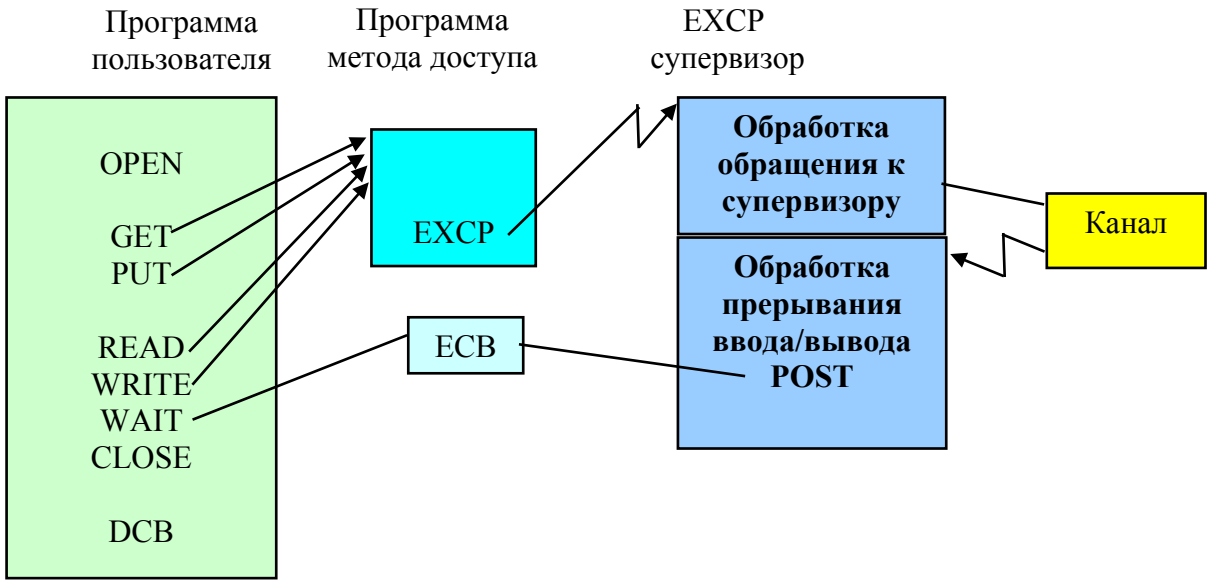


Рис. 89. Схема работы метода доступа

С помощью макрокоманд происходит передача управления соответствующей программе метода доступа. После выполнения определенных действий

метод доступа вызывает EXCP супервизор с помощью программного прерывания. Супервизор запускает канал.

После завершения работы канал инициирует прерывание ввода/вывода, которое обрабатывает другая часть супервизора. После обработки супервизор выдает макрокоманду POST и инициирует работу метода доступа.

Разумеется, работа метода доступа зависит от способа доступа. Если метод доступа с очередями, то в нем выполняются управление буферами, блокирование и разблокирование записей, синхронизация с супервизором ввода/вывода, обработка ошибок. В случае базисного метода доступа все эти действия реализует программист в коде программы.

10.1.4 Буферизация в z/OS

Буферизация используется в методах доступа с очередями. Конечно, программист может использовать буфера и в других случаях, но тогда придется в программе реализовывать управление самостоятельно.

Для реализации буферизации ввода/вывода создается буферная область – область ОП , предназначенная для временного хранения входных данных, считываемых в память с внешнего устройства, или выходных данных, подготавлеемых для пересылки на внешнее устройство.

Буфер – часть буферной области, выделенной для одной операции обмена. Совокупность связанных в цепочку буферов состовляют буферный пул. В начале буферного пула находится блок управления буферным пулом. Буфер используется для компенсации различий в скоростях работы физических устройств. При обмене данными в буфер помещается блок логических записей, соответствующий физической записи на внешнем устройстве.

Получение буферной области осуществляется различным способом. Можно просто получить область ОП с помощью макрокоманд управления памятью GETMAIN и FREEMAIN. В этом случае память может отводится в области задачи. Также существуют специальные макрокоманды GETPOOL и FREEPOOL. В этом случае память отводится в системной области памяти. Память под буферный пул может отводится автоматически методами доступа с очередями. В этом случае существует возможность задать число буферов в параметрах макрокоманды обращения к методу доступа.

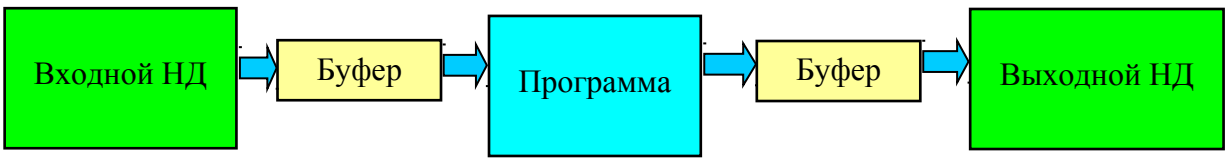


Рис. 90. Схема использования буферов

На рис. 90 входному НД соответствуют входные буфера, а выходному НД выходные буфера. Передача данных из входного НД в буфер называется загрузкой, а передача данных из буфера в выходной НД – разгрузкой. Передача между программой и буфером называется пересылкой.

Различают буферизацию с упреждением и буферизацию по требованию. Буферизация с упреждением используется в методах доступа с очередями. В

этом случае записи обрабатываются последовательно и известен адрес следующей записи. Это позволяет еще до запроса данных в программе загрузить входные буфера блоками логических записей. При загрузке входные буфера заполняются данными при выполнении макрокоманды OPEN. Затем данные подгружаются по мере освобождения очередного буфера. По макрокоманде GET программа получает доступ к очередной логической записи. В блоке управления буферным пулом отслеживается положение текущего буфера.

Данные из выходных буферов разгружаются по мере формирования блоков логических записей. По мере вывода из программы логических записей по макрокоманде PUT формируется блок и заполненный буфер выводится в физическую запись.

Буферизация по требованию используется в базисных методах доступа. В этом случае записи обрабатываются в произвольном порядке. Поэтому адрес следующей записи неизвестен и определяется только в момент запроса. Следовательно блок логических записей загружается в буфер в момент запроса. При чтении запрос определяется макрокомандой READ. После выдачи запроса работа программы продолжается поскольку метод доступа не осуществляет синхронизацию. Поэтому должна быть использована макрокоманда WAIT в подходящем месте. После завершения операции ввода в буфере находится блок логических записей. Выделение логических записей осуществляет программа.

При выводе программа формирует в буфере блок логических записей и макрокомандой WRITE этот блок пишется на внешнее устройство. Контроль операции вывода и синхронизация выполняются в программе.

Пересылки осуществляются методом доступа только в случае использования методов доступа с очередями. В параметрах макрокоманд указывается режим пересылки. Существуют следующие режимы.

Режим перемещения (рис.91) требует выделения в программе рабочей области памяти. По макрокоманде GET логическая запись из буфера копируется в рабочую область. По макрокоманде PUT логическая запись из рабочей области копируется в буфер. В макрокомандах указывается адрес рабочей области.

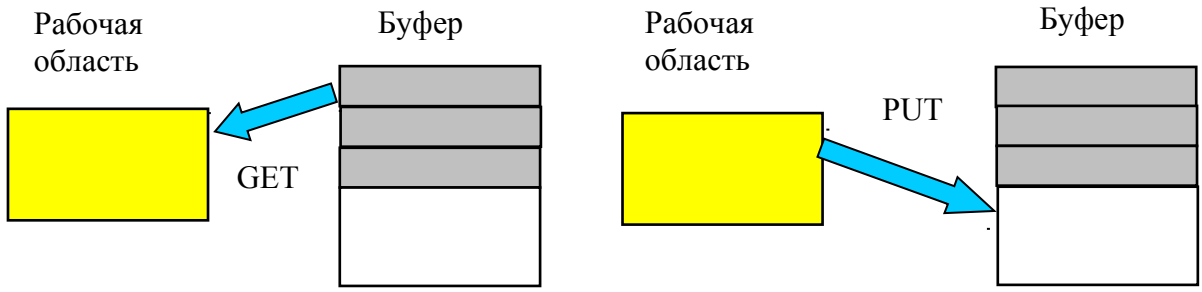


Рис. 91. Пересылка в режиме перемещения

Пересылка в режиме указания (рис.92) не требует рабочей области. По макрокоманде GET в регистр 1 пересылается адрес логической записи, а по макрокоманде PUT адрес с которого должна быть помещена логическая запись. Это позволяет обрабатывать запись во входном буфере, но в выходной

буфер запись нужно пересылать. На рис. 94 показана пересылка в режиме указания.

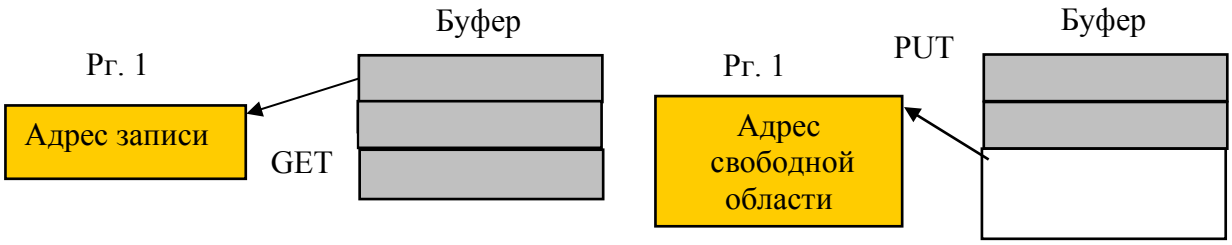


Рис. 92. Пересылка в режиме указания

Пересылка в режиме подстановки (рис. 93) используется в том случае, когда характеристики входного и выходного НД совпадают. То есть они имеют одинаковые длину записи, размер блока, формат логической записи.

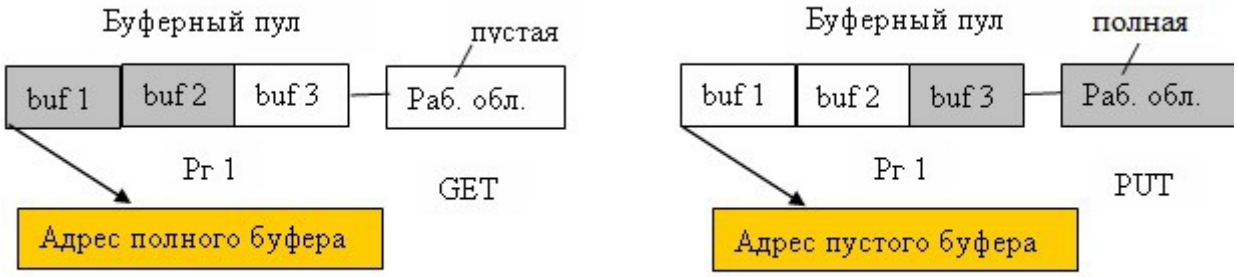


Рис. 93. Пересылка в режиме подстановки

Макрокоманда GET в режиме подстановки добавляет пустую рабочую область к буферному пулу и в регистр 1 помещает адрес заполненного буфера, а макрокоманда PUT добавляет заполненную рабочую область к буферному пулу и в регистр 1 помещает адрес пустого буфера. Рабочая область в этом случае должна иметь такой же размер как буфер в буферном пуле.

Простая и обменная буферизации

В зависимости от режимов пересылок различают простую и обменную буферизации. Простая буферизация использует режимы пересылок перемещения и указания. Она используется в случае различных по своим характеристикам НД. Например, один НД содержит записи фиксированной длины, а другой переменной. При использовании простой буферизации для входного и выходного НД можно использовать различные пары режимов пересылок.

Пусть для входного и выходного файлов используется режим перемещения. В этом случае используется рабочая область, в которой производится обработка записи (рис. 94). Макрокоманды GET и PUT работают в режиме перемещения.

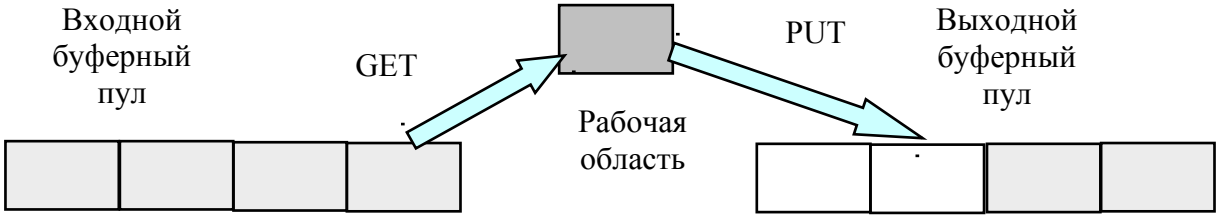


Рис. 94. GET и PUT в режиме перемещения

По GET содержимое заполненного буфера переносится в рабочую область, где запись обрабатывается. По PUT запись из рабочей области помещается в свободный буфер выходного пула.

Возможно использовать для входного буферного пула макрокоманду GET в режиме перемещения, а для выходного пула макрокоманду PUT в режиме указания. На рис. 95 показана эта комбинация режимов пересылки.

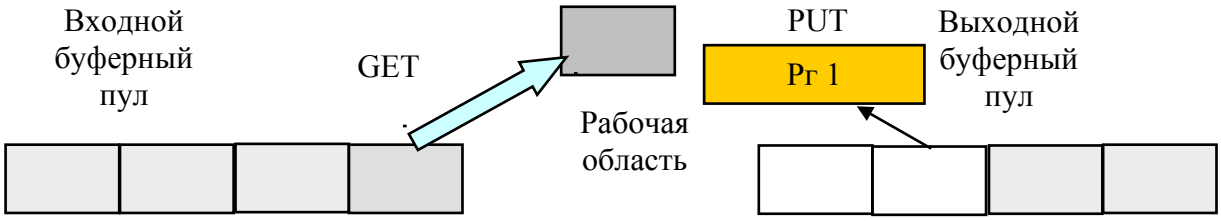


Рис. 95. GET в режиме перемещения, PUT в режиме указания

В этом случае по макрокоманде GET в рабочую область переносится запись. Обработка ведется в рабочей области, а затем запись перемещается в свободный буфер по адресу в регистре 1, полученному макрокомандой PUT.

Также можно рассмотреть комбинацию использования для входного буферного пула макрокоманды GET в режиме указания, а для выходного буферного пула макрокоманды PUT в режиме перемещения (рис. 96).

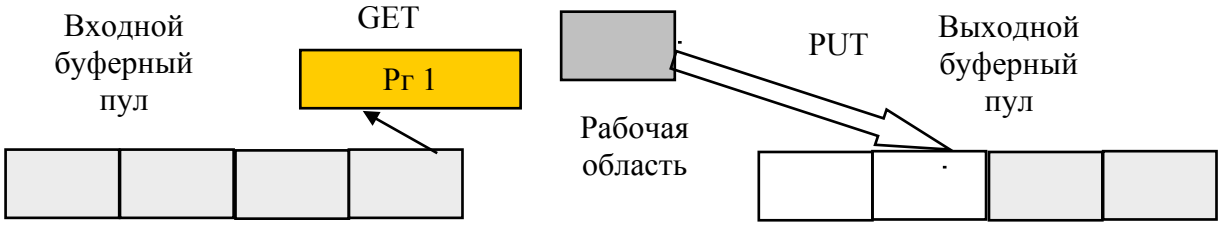


Рис. 96. GET в режиме указания, PUT в режиме перемещения

В этом случае в регистр 1 макрокомандой GET помещается адрес входной записи. Она может обрабатываться в буфере, а затем переписывается в рабочую область, где форматируется и перемещается в выходной пул макрокомандой PUT.

И наконец, последняя возможная комбинация режимов пересылок, когда для входного и выходного буферных пулов используются макрокоманды в режиме указания (рис. 97).

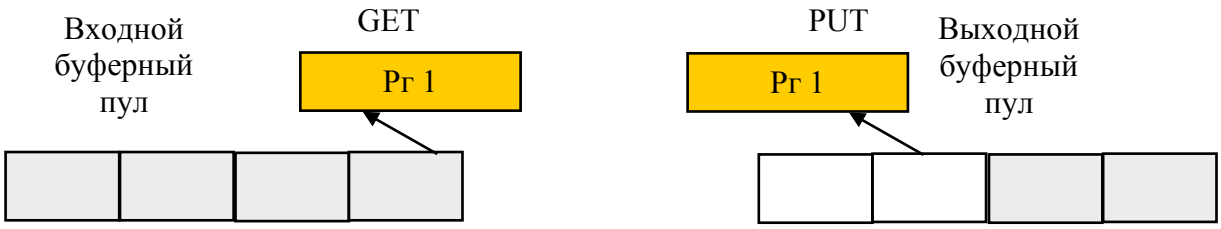


Рис. 97. GET в режиме указания, PUT в режиме указания

В этом случае и макрокоманда GET и макрокоманда PUT записывают адреса входной записи и свободного буфера в регистр 1. Разумеется, в программе после первой выполненной макрокоманды значение регистра 1 должно быть сохранено поскольку следующая макрокоманда изменяет его. Обработка ведется в буфере и затем перемещается из входного буфера в выходной программой.

Обменная буферизация используется в случае, когда входной и выходной НД имеют одинаковую структуру. Например, когда необходимо скопировать НД с одного устройства на другое. В этой ситуации используется режим подстановка (рис. 98).

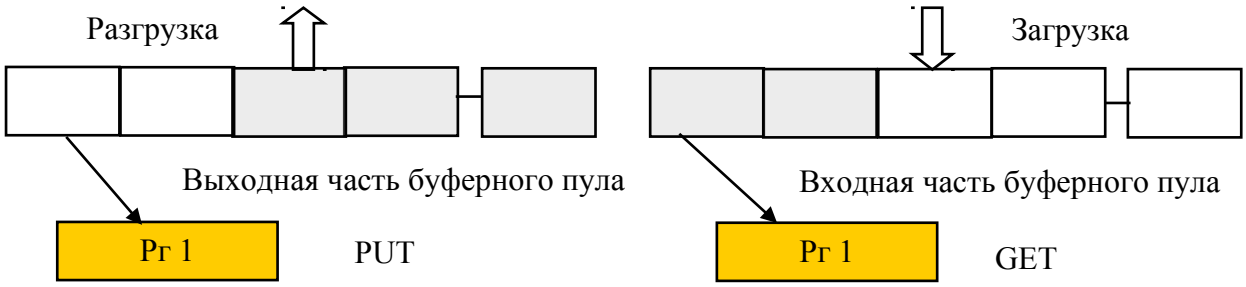


Рис. 98. GET и PUT в режиме подстановки

Макрокоманда GET добавляет свободный буфер к буферному пулу и в регистр 1 помещает адрес полного буфера. Макрокоманда PUT добавляет к буферному пулу заполненный буфер и помещает в регистр 1 адрес пустого буфера. Таким образом можно использовать один буферный пул, а в программе получить адрес свободного буфера по PUT и подставить его в качестве параметра в GET. Полученный адрес полного буфера подставить в качестве параметра в PUT. Выполняя в цикле макрокоманды GET и PUT можно копировать НД. В этом случае макрокоманды должны использовать разные адреса DCB, которые связаны с разными НД.

В заключение следует отметить, что в z/OS различные методы буферизации помогают программистам создавать более эффективные программы. В других системах с прямым вводом/выводом подобные механизмы недоступны программистам. Буферизация реализована в системных программах.

10.2 Реализация функций управления вводом/выводом в UNIX

В системах с прямым вводом/выводом обмен данными с внешним устройством осуществляется драйвером устройства. Драйвер устройства поставляется производителем устройства и для стандартных устройств, таких

как клавиатура, USB и Bluetooth устройства, содержатся в дистрибутиве ОС. Для других устройств, таких как принтер, драйверы поставляются на оптическом диске или скачиваются из Интернета.

10.2.1. Драйвер устройства

Драйвер является программой выполняющейся на ЦП и соответственно занимающей время на обмен. Драйвер необходимо установить в системе, то есть система ввода/вывода должна иметь возможность найти этот драйвер для устройства и инициировать его работу.

Типичный драйвер устройства содержит , как минимум, три основных блока. Это заголовок драйвера, блок стратегии и блок прерываний. Заголовок содержит информацию о драйвере и об устройстве, такую как имя и тип устройства, число однотипных устройств, обслуживаемых одновременно. Заголовок также содержит адреса блоков стратегии и прерываний. Блок стратегий принимает запросы на выполнение операций, ведет очередь этих запросов, осуществляет запуск операции обмена и обрабатывает ее завершение. Запрос на выполнение операции содержит код операции, адрес буфера ОП и адрес области на устройстве в случае накопителя на диске. Для других устройств формат запроса будет другой. В любом случае запрос содержит информацию необходимую для выполнения операции. Блок прерываний выполняет собственно обмен с контроллером устройства, посылая команды и обрабатывая ответы. Работа блока прерываний подобна работе канальной программы, но с той разницей что выполняется на ЦП, а не на периферийном процессоре.

Блок прерываний посылает команду контроллеру устройства и переходит в состояние ожидания. Контроллер устройства завершив операцию возбуждает прерывание. Это прерывание передает управление блоку прерываний, который посылает следующую команду. Когда последовательность команд выполнена и заданная операция завершена, управление передается блоку стратегий.

В современных ОС в связи с появлением новых интерфейсов и увеличением их скорости передачи используются многоуровневые драйверы. Например, устройства печати теперь могут использовать USB вместо параллельного интерфейса. В этом случае драйвер сложного устройства готовит информацию, а для передачи обращается к драйверу USB.

10.2.2. Поиск драйвера устройства

Другой вопрос связан с поиском нужного драйвера, который определяется по типу устройства. Устройство для ввода/вывода определяется файловой системой, которая отыскивает нужные данные на определенном устройстве. Существуют также стандартные устройства ввода и вывода, которые заранее предопределены в системе. Конечно эти устройства могут быть переопределены и стандартный ввод/вывод может быть перенаправлен. В этом случае также известны эти устройства. Наконец в прикладной программе может

быть определено устройство для обмена информацией. Таким образом устройство определяется и передается системе ввода/вывода.

Теперь нужно для данного устройства найти драйвер. В ранних ОС, использовавших прямой ввод/вывод, адрес драйвера указывался в UCB при установке, а сам драйвер устанавливался резидентно. Такой способ позволяет быстро найти драйвер, но система становится жестко привязана к аппаратной платформе. Другой подход использовался в системе UNIX. Описания драйверов содержатся в файловой системе в виде специальных файлов в каталоге DEV и в подкаталогах (рис. 99).

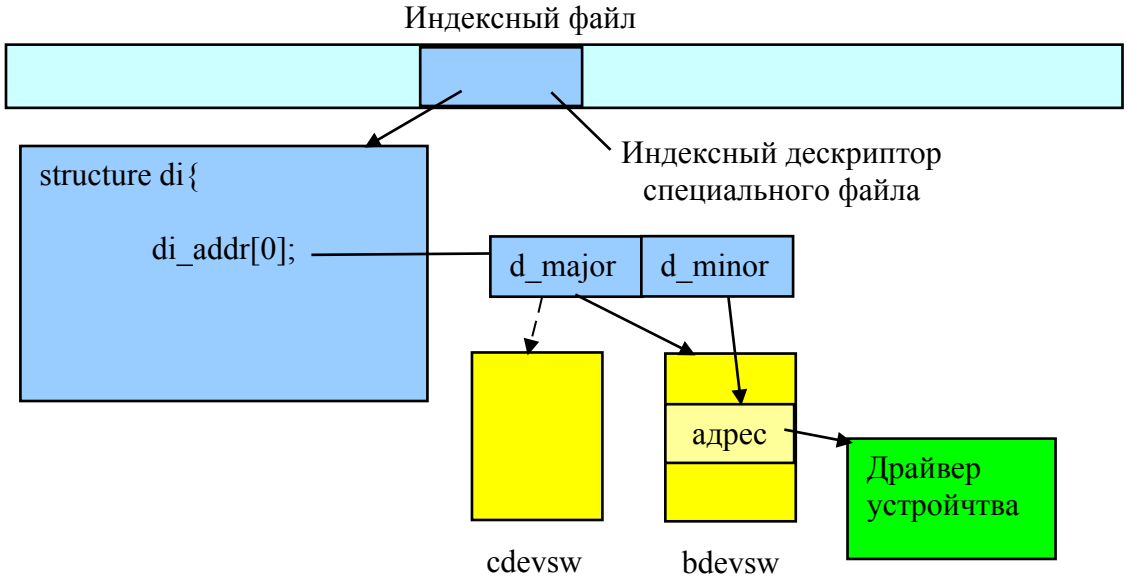


Рис. 99. Поиск драйвера по индексному дескриптору

Специальные файлы не хранят данные и являются фиктивными файлами. Они обеспечивают механизм отображения физических внешних устройств в имена файлов файловой системы. Каждому устройству, поддерживаемому системой, соответствует, по меньшей мере, один специальный файл. Специальные файлы создаются при выполнении системного вызова mknod, каждому специальный файл соответствует драйверу соответствующего устройства. При выполнении чтения или записи производится прямой вызов соответствующего драйвера, программный код которого отвечает за передачу данных между процессом пользователя и соответствующим физическим устройством.

Как любые другие файлы, специальные файлы имеют индексный дескриптор в системном индексном файле. Индексный дескриптор файла представляет собой структуру с полем di_mode, в котором тип файла показывает устройство блоковое или символьное, и di_addr[0], которое состоит из двух полей: d_major и d_minor, которые определяют тип и номер устройства. Тип устройства может быть символьный (байт-ориентированные устройства) или блочное (блок-ориентированное устройства). В различных версиях UNIX и UNIX-подобных системах вводят и другие типы, Номер устройства передается драйверу как параметр. Таким образом для определения драйвера отыскивается соответствующий индексный дескриптор специального файла в ин-

дексном файле, по типу устройства определяется таблица cdevsw для символьных устройств или bdevsw для блочных устройств. По номеру устройства в таблице находится драйвер.

10.2.3. Организация ввода/вывода

Блочные специальные файлы ассоциируются с такими внешними устройствами, обмен с которыми производится блоками байтов данных, размером 512, 1024, 4096 или 8192 байтов. Типичным примером подобных устройств являются магнитные диски.

Символьные специальные файлы ассоциируются с внешними устройствами, которые не обязательно требуют обмена блоками данных равного размера. Примерами таких устройств являются терминалы (в том числе, системная консоль), последовательные устройства, некоторые виды магнитных лент. Иногда символьные специальные файлы ассоциируются с магнитными дисками.

При обмене данными с блочным устройством система буферизует данные во внутреннем системном кеше. Через определенные интервалы времени система "выталкивает" буфера, при которых содержится метка "измененный". Кроме того, существуют системные вызовы sync и fsync, которые могут использоваться в пользовательских программах, и выполнение которых приводит к выталкиванию измененных буферов из общесистемного пула. Основная проблема состоит в том, что при аварийной остановке компьютера (например, при внезапном выключении электрического питания) содержимое системного кеша может быть утрачено. Тогда внешние блочные файлы могут оказаться в рассогласованном состоянии. Например, может быть не вытолкнут супер-блок файловой системы, хотя файловая система соответствует его вытолкнутому состоянию. Заметим, что в любом случае согласованное состояние файловой системы может быть восстановлено (конечно, не всегда без потерь пользовательской информации). В современных системах эта проблема решается аппаратными средствами

Обмены с устройствами, определяемыми символьными специальными файлами производятся напрямую, без использования системной буферизации.

При создании процесса в его адресном пространстве резервируется область памяти называемая контекстом, в котором создается структура U. В поле u_file записывается указатель на активный дескриптор файла file, который располагается в системной области памяти и строится при первом открытии этого файла каким-либо процессом. В дескрипторе file содержится указатель на индексный дескриптор этого файла в индексном файле. Также там содержится признак типа открытия файла на чтение, запись или как программный канал, счетчик числа ссылок к дескриптору, который показывает число процессов, открывших файл.

При выполнении системных вызовов read и write аргументы этих функций и информация из дескриптора файла используются для установки переменных в области контекста. Это поле u_base с адресом буфера ввода или вы-

вода в основной памяти, поле u_count как счетчик числа байт, поле u_offset для указания позиции в файле и для блочных устройств содержит адрес очередного блока (рис. 100).

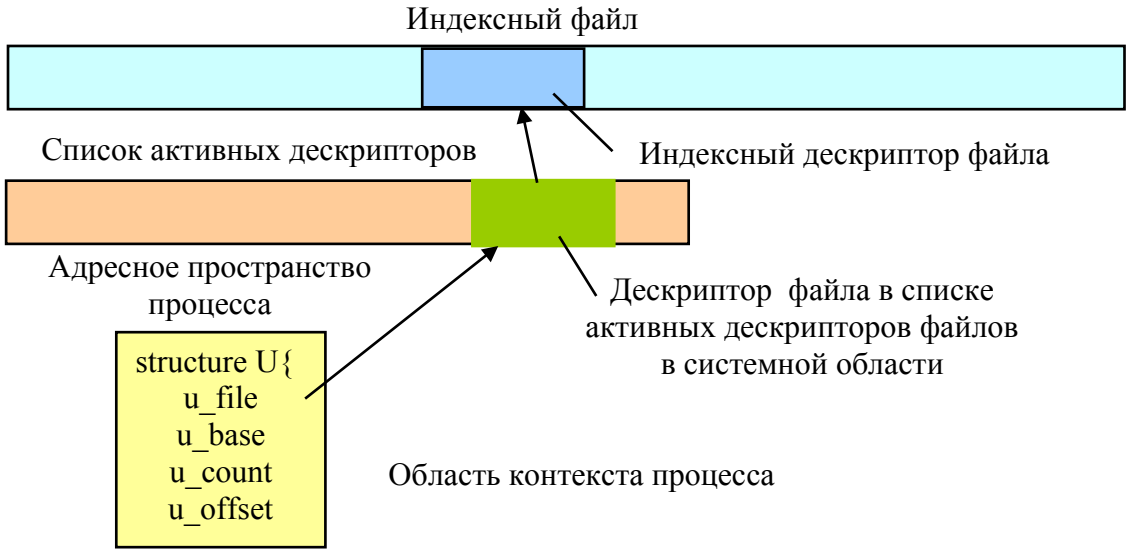


Рис. 100. Организация ввода/вывода

Описанная организация ввода/вывода не зависит от особенностей аппаратуры, но поиск драйвера через файловую систему является медленным. Поэтому в системе используется глобальная буферизация, когда дескрипторы файлов и блоки открытых файлов хранятся в системном кэше.

Тема 11. УПРАВЛЕНИЕ ДАННЫМИ В ОС

Компьютерные системы предназначены для хранения и обработки информации. Функции хранения и управления информацией требуют организации данных и должны обеспечивать эффективный доступ к информации. Также необходимо обеспечивать разграничения полномочий доступа и защиту информации, надежное хранение данных и восстановление в случае сбоев. Система управления данными позволяет изолировать пользователей от машинно зависимых объектов и обеспечить централизованный доступ к внешней памяти.

Часть этих функций выполняет файловая система, а часть обеспечивается другими средствами. Так разграничение полномочий доступа обеспечивает система управления данными ОС, а надежное хранение обеспечивают специальные системы внешней памяти. В этом разделе рассматриваются средства ОС управления данными.

11.1 Логические записи и блокирование

Единицей адресуемой памяти в современных системах является байт. Последовательность байтов есть запись. Различают логические и физические записи. С логическими записями работает программа и функция чтения предоставляет программе логическую запись. Программа предоставляет функции вывода также логические записи, которые объединяются в блоки и пишутся на устройство как физическая запись. Для FBA устройств размер физической записи 512 байт равен размеру физического блока.

Используются следующие типы логических записей. Записи фиксированной длины (тип F). Такая запись содержит фиксированное число байт. Записи переменной длины (тип V) должны иметь либо счетчик в начале записи, содержащий количество байт в записи, либо некоторый символ в конце записи (EOL), означающий окончание записи. Первый тип используется в z/OS, а символ в конце записи (00h) в системе программирования C/C++. Существуют также в z/OS записи неопределенной длины. Они используются для представления загрузочных модулей. В этом случае длина записи равна длине файла.

Однако писать и читать на диск логические записи неэффективно. Большое время тратится на подвод головки, а не на саму операцию чтения или записи. Обмен с устройством эффективнее вести большими порциями информации. Для этого логические записи объединяют в блоки. А блоки пишутся в виде физических записей. При использовании FBA устройств, в которых размер блока небольшой (512 байт), информация пишется кластерами по нескольку блоков в кластере.

Блокирование записей фиксированной длины выполняется в блоки, размер которых кратен размеру записи. Поэтому управляющей информации не требуется. При блокировании записей переменной длины в начале блока со-

держится счетчик с числом байт, записанных в блок (рис. 101). Записи неопределенной длины не блокируются

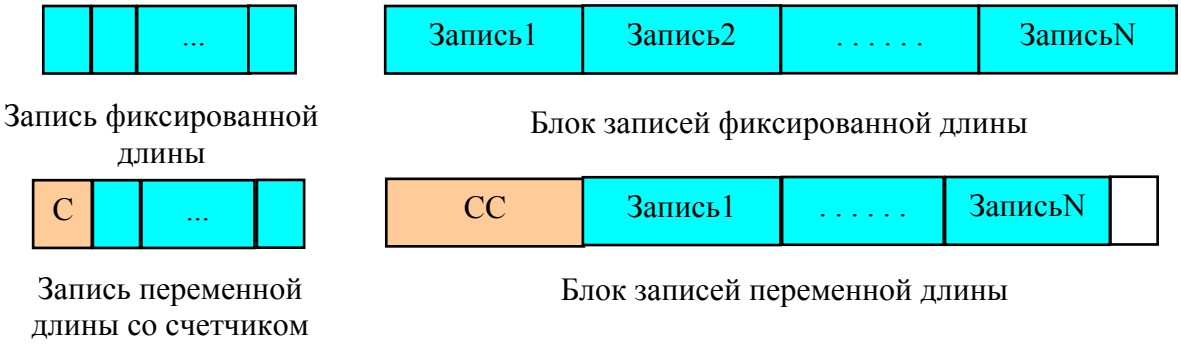


Рис. 101. Форматы логических записей

Таким образом система управления данными имеет дело как с логическими записями, так и с физическими записями или блоками логических записей.

11.2 Файлы и наборы данных

Файл или набор данных представляет собой поименованную совокупность данных, объединенных общим назначением и имеющие определенную организацию.

Система управления данными позволяет реализовать принцип независимости программ от характеристик данных и внешних устройств. Это означает, что программист не должен указывать физические адреса на диске где располагаются данные, размер и структуру данных. Указывается только имя и общие атрибуты. Следует заметить, что в современных компьютерах программировать в физических адресах и невозможно. Сложные аппаратные системы ввода/вывода, мультипрограммный режим работы, при котором несколько процессов одновременно могут вести обмен информацией с одним устройством, разграничение доступа к данным между пользователями требуют централизованной системы управления данными.

Данные могут иметь определенную структуру и интерпретацию. С этой точки зрения файлы рассматриваются ОС как неструктурированная и неинтерпретированная информация. Это просто поименованная последовательность байт с прямым доступом. НД в отличие от файлов структурированы, но содержат не интерпретированную информацию. Так НД могут быть последовательными, библиотечными, прямого доступа, расширенного библиотечного формата, VSAM. Структурированную и интерпретированную информацию содержат базы данных, организованные некоторой системой управления базой данных (СУБД). Так в реляционных СУБД данные хранятся в таблицах, а для каждого данного известно, что это есть имя, денежная единица, номер телефона, табельный номер служащего и так далее.

Файлы и НД бывают временными и постоянными. Постоянные остаются на диске после окончания процесса или задания и уничтожаются по специальному приказу. Временные существуют только во время работы процесса

или задания. Они создаются в процессе выполнения и уничтожаются после окончания системой. Так, например, если в диспозиции НД второй параметр имеет значение PASS для всех шагов в задании, то после окончания задания система уничтожит этот НД.

НД в z/OS могут быть каталогизированными и некаталогизированными. Для каталогизации НД используется специальная утилита. В Open System все файлы каталогизированы.

11.3 Функции системы управления данными

Система управления данными выполняет следующие функции.

1) Учет местонахождения информации в системе.
2) Размещение информации.
3) Выделение информационного ресурса.
4) Освобождение информационного ресурса.

11.3.1 Учет местонахождения информации в системе

Необходимо учитывать не только созданные и хранящиеся в системе файлы, но и учитывать свободные области памяти на носителях. В различных ОС это выполняется разным способом.

В z/OS для поиска НД на томе используется VTOC, в котором хранятся DSCB. Чтобы найти том с НД используется каталог. После того как том найден, во второй записи (считая с 0) на дорожке 0 цилиндре 0 находится адрес VTOC, который может располагаться в пределах 64К цилиндров. После того, как VTOC найден, в нем отыскивается DSCB НД. Для более быстрого поиска НД используется индексный VTOCX. Свободное пространство на томе также учитывается с помощью DSCB специального типа.

В ОС Windows используется файловая система NTFS. Эта популярная система устроена следующим образом. Она делит все доступное для нее место на диске на кластеры. Размер кластера может быть от 512 байт до 64Кбайт, но наиболее популярный размер 4Кбайта. Дисковое пространство делится на следующие части. 12% отводится для MFT (master file table), а остальное пространство отводится под файлы (рис. 102).

Рис. 102. Структура диска под NTFS

Зона MFT пуста и служит для расширения таблицы при увеличении числа файлов. Однако, если файлы переполняют отведенное им пространство, то зона сокращается в два раза. Таким образом система оптимизирует использование памяти. Если файлов много и они небольшие, то MFT увеличивается и занимает свободную память в зоне. Если преобладают файлы большого размера, то часть свободной зоны занимают файлы. Часть таблицы с наиболее важными файлами дублируется в середине диска.

Структура MFT это последовательность элементов таблицы. Каждый элемент содержит описание файла и представляет собой запись фиксированной длины (1 Кбайт). Каждая запись соответствует какому-либо файлу. В ней хранится имя файла, размер, адрес на диске и другая информация. Если для информации не хватает одной записи MFT, то используются несколько, причем не обязательно подряд. Первые 16 файлов носят служебный характер и недоступны другим программам. Они называются метафайлами, причем самый первый метафайл это сама MFT. Эти первые 16 элементов MFT – единственная часть диска, имеющая фиксированное положение. Интересно, что вторая копия первых трех записей, для надежности хранится ровно посередине диска.

Метафайлы содержат системную информацию и находятся в корневом каталоге. Их имена начинаются с символа '&'. Прежде всего это сам файл &MFT – сама таблица.

&MFTmin – резервная копия первых 16 записей посередине диска.

$LogFile – файл для журналирования, в который записываются текущие изменения при работе файловой системы. Это позволяет восстановить систему в случае сбоя.

$Volume – метка тома.

$AttrDef – описание атрибутов файлов.

$. – файл корневого каталога.

$Bitmap – карта свободных и занятых кластеров для учета свободного пространства.

$Boot – загрузочный сектор.

$Quota – полномочия пользователей для доступа к файлам.

$Upcase – соответствие заглавных и прописных букв в именах файлов. Имена файлов представлены в кодировке Unicode с двухбайтовым представлением и поиск букв требует временных затрат.

Файлы могут не иметь представления в области для файлов. Это происходит в случае, если файл не имеет данных или он небольшого размера. В этом случае данные помещаются в MFT. Единственным представлением файла является запись в MFT.

Каждый файл имеет некоторое абстрактное строение. Он состоит из потоков данных. Один из потоков и содержит данные файла. Но большинство атрибутов файла тоже являются потоками.

Каталог на NTFS представляет собой специфический файл, хранящий ссылки на другие файлы и каталоги, создавая иерархическую файловую структуру на диске. Файл каталога поделен на блоки, каждый из которых содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который уже предоставляет полную информацию об элементе каталога.

В ОС UNIX структура размещения файловой системы на диске показана на рис. 103. Первый блок содержит программу начальной загрузки IPL (initial program load). Эта программа загружает в конец памяти загрузчик ОС.

Рис. 103. Структура тома в UNIX

Суперблок содержит важную информацию необходимую для функционирования системы и представляет собой структуру s. Он содержит следующие поля:

s_type – тип файловой системы.

s_fsize – номер блока до которого размещается файловая система. Со следующего блока начинается область выгрузки.

s_free[NICFREE] – массив номеров свободных блоков. В s_free[0] содержится адрес блока с номерами свободных блоков. Поскольку число свободных блоков большое и не может поместиться в массиве, то их адреса хранятся в других блоках и по мере необходимости подкачиваются в массив.

s_nfree – индекс массива. Когда массив заполнен, s_nfree= NICFREE-1. При выделении свободных блоков индекс s_nfree уменьшается до 0. Когда индекс становится равным 0, то осуществляется подкачка в массив номеров свободных блоков и индекс устанавливается на последний элемент.

s_isize – размер индексного файла.

s_inode[NICINODE] – массив адресов свободных индексных дескрипторов в индексном файле.

s_ninode – индекс в массиве s_inode. Когда массив заполнен, s_ninode= NICINODE-1. При создании файла предоставляется свободный индексный дескриптор и индекс уменьшается. Когда все свободные индексные дескрипторы в массиве исчерпаны, сканируется индексный файл и из него выбираются свободные дескрипторы и их адреса помещаются в массив. Индекс указывает на последний элемент массива с адресом свободного индексного дескриптора.

Таким образом при создании файла из суперблока выбирается свободный дескриптор в индексном файле, в котором создается индексный дескриптор файла, и свободные блоки для данных файла.

s_flock – флаг запрещающий работу со списком свободных блоков.

s_ilock – флаг запрещающий работу со свободными индексными дескрипторами.

s_fmod – флаг модификации суперблока. Если он модифицировался, то необходимо обновить копию на диске.

s_ronly – файловая система монтируется только для чтения.

s_time – время последней модификации суперблока.

Индексный файл представляет собой массив структур dinode. Каждый дескриптор имеет следующие поля.

di_mode – тип файла, права доступа и атрибуты (рис. 104).

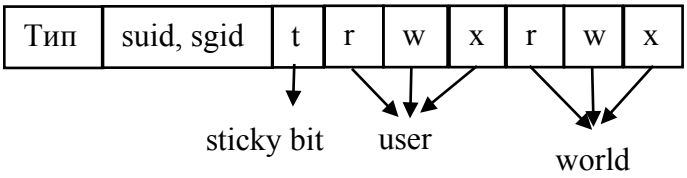


Рис. 104. Формат поля di_mode

Тип файла может быть как просто файл IFREG, каталог IFDIR и в специальных файлах IFBLK – блочное устройство, IFCHR – символьное устройство. Sticky bit используется в файлах каталогах для установки ограничения на возможность удаления файлов из каталога только для владельцев этого файла. Поля suid и sgid содержат идентификатор системного пользователя и идентификатор системной группы. Группы флагов r, w и x для владельца файла и для остальных пользователей задают полномочия доступа только по чтению, по записи и выполнению файла.

di_nlink – число ссылок на дескриптор из каталогов. Если 0, то гнездо дескриптора свободно.

di_uid, di_gid – код идентификации пользователя.

di_size – длина файла.

di_atime – время последнего обращения к файлу.

di_mtime – время создания или последней модификации файла.

di_addr – массив из 13 элементов. Первые 10 элементов содержат адреса блоков файла. Остальные содержат адреса косвенной адресации (рис.105). Первые 10 блоков файла адресуются непосредственно из элементов массива di_addr[0] до di_addr[9]. Элемент di_addr[10] адресует блок с адресами блоков файла. Блок с адресами содержит 256 элементов. Элемент di_addr[11] адресует блок с адресами, которые указывают также на блоки с адресами, а те уже указывают на блоки файла. И, наконец, элемент di_addr[12] указывает на блок с адресами блоков, которые содержат адреса блоков, которые в свою очередь также содержат адреса блоков файла.

Таким образом к файлам небольшого размера доступ осуществляется быстро. Но чем больше размер файла, тем большее время требуется для доступа к его блокам. Кроме того, читать каждый раз суперблок, индексный дескриптор файла и дескрипторы открытых файлов занимает много времени

и существенно замедляет быстродействие системы. Поэтому суперблок, индексные дескрипторы часто используемых файлов и другая информация считывается в системный кэш. Подвергаются кэшированию и блоки открытых файлов.

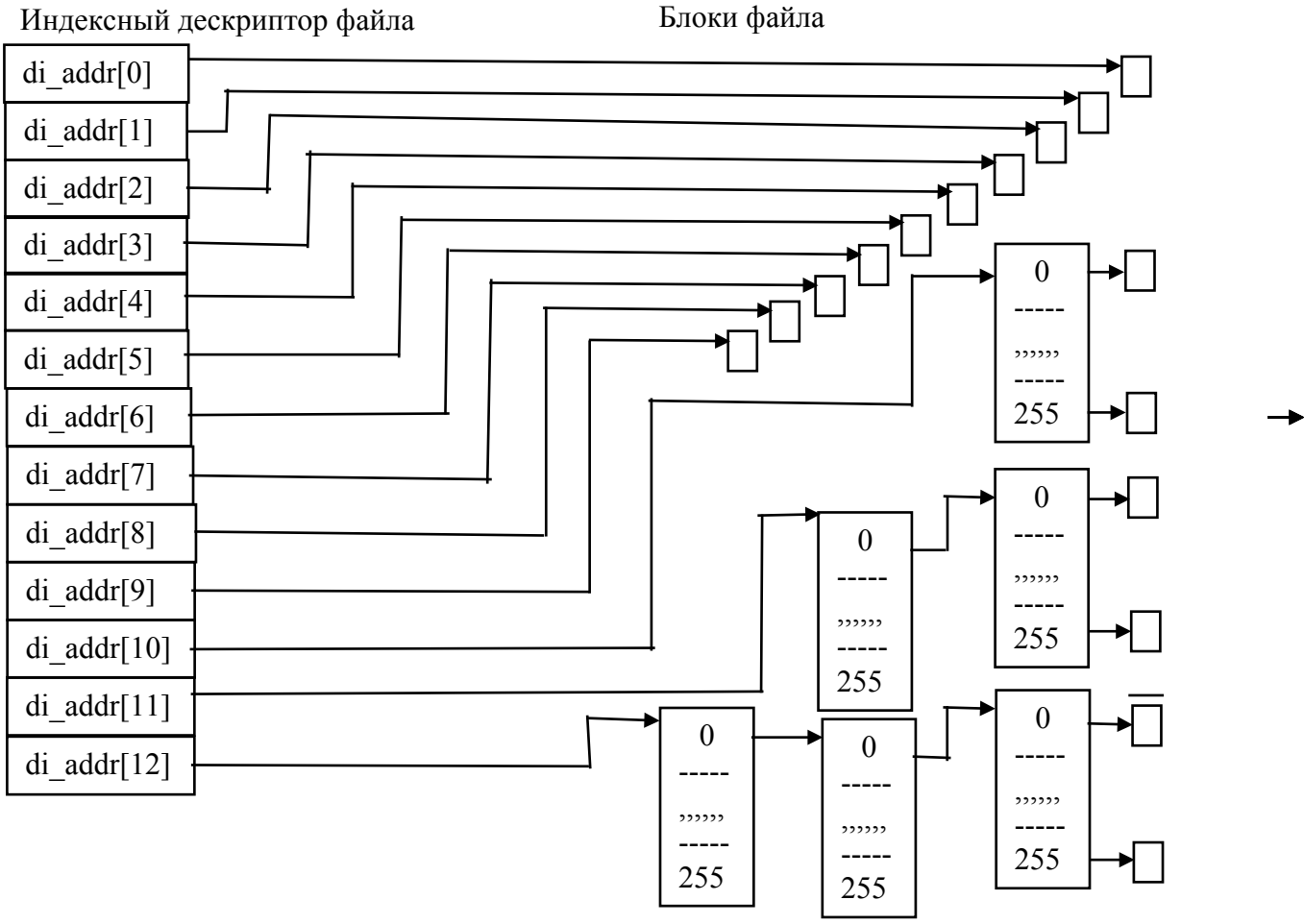


Рис. 105. Хранение информации о файле

Но такое решение предъявляет повышенные требования к энергобезопасности системы. Внезапное отключение питания разрушало файловую систему и это являлось огромной проблемой для пользователей UNIX. В настоящее время эта проблема решена благодаря надежным источникам бесперебойного питания.

11.3.2 Размещение информации в системе

КС содержит разнообразную информацию. Прежде всего это системная информация, которую составляют образ ядра и обеспечивающие программы, такие как службы или демоны. Эти программы инициируются в момент загрузки системы. Затем в систему устанавливаются прикладные программы. Это различные текстовые процессоры, мультимедийные приложения, системы программирования и другие пакеты. И, наконец, данные пользователей.

Все эти программы размещаются в своих каталогах и доступ к ним возможен только пользователям определенных категорий. В качестве примера можно рассмотреть клиентские системы Windows. Системные программы хранятся в каталоге Windows, устанавливаемые программы ставятся в каталог Program Files, а для разных пользователей в каталоге Users создаются каталоги для пользователей.

Эти данные должны быть отделены друг от друга и к некоторым данным доступ определенных категорий пользователей может быть закрыт. Это достигается в системе с помощью механизма назначения полномочий доступа.

11.3.3 Механизм разграничения полномочий доступа в системе

Во многих системах используется механизм разграничения полномочий доступа к ресурсам на основе кода идентификации пользователя UIC (user identification code). UIC представляет собой пару целых чисел без знака обычно в диапазоне от 0 до 255. Например, [5,23], где первое число есть номер группы, а второе – номер члена группы.

UIC создается при создании учетной записи пользователя в системе вместе с именем и паролем. В дальнейшем при входе в систему UIC присваивается пользователю, то есть при создании им файла, каталога, этим ресурсам присваивается UIC. Кроме того при запуске приложения или утилиты этот код имеет запущенная программа.

При создании ресурса он получает не только UIC, но и код, который определяет для разных категорий пользователей возможности владения этим ресурсом, возможные действия. Пусть возможные действия будут следующие: чтение (r), запись (w), изменение (e), удаление (d). Категории пользователей могут быть такие, как системный, член группы, владелец ресурса и все другие. Системный пользователь как правило имеет фиксированный UIC равный [255,255] или [0,0].

Проверка полномочий доступа происходит следующим образом. Пусть файл имеет UIC = [x,y] и полномочия для системного пользователя r---. Это значит, что он может только читать этот файл. Член группы имеет полномочия rw--. владелец – rwed и все другие – ----. Также пусть программа, которая собирается открыть этот файл на запись, имеет UIC = [z,w]. Первое, что происходит, это проверка UIC программы на предмет, является ли владелец программы системным пользователем. Если это так, то функция open возвращает ненулевой код возврата и открытие файла не производится. Если это не системный пользователь, то сравниваются UIC программы и UIC файла. Если $x \neq z$ и $y \neq w$, то это любой другой пользователь и файл не открывается. Если

x=z и y≠w, то это член группы, или x=z и y=w, то это владелец файла, и файл будет открыт на запись.

Такой механизм реализован в UNIX с тем замечанием, что UIC системного пользователя записан в di_mode дескриптора файла, а возможные действия ограничены чтением, записью и исполнением загрузочного модуля.

11.3.4 Выделение и освобождение информационного ресурса

Выделение информационного ресурса состоит в предоставлении свободных блоков при создании информационного ресурса и в предоставлении данных в существующем информационном ресурсе.

В z/OS выделение свободных экстентов в томе осуществляется путем поиска DSCB, содержащих адреса и размер свободных участков. В NTFS для получения номеров свободных блоков используется файл BITMAP, в котором биты соответствуют блокам и значение бита говорит о том, свободен блок или занят. В UNIX номера свободных блоков берутся из массива в суперблоке.

Доступ к файлу не столь тривиален. Файловая система должна найти физические адреса блоков файла и предоставить их системе ввода/вывода, которая совершает обмен с устройством. Кроме того должна проверять полномочия доступа к файлу и разделение файла между несколькими процессами, если эти процессы открывают файл на чтение. Это осуществляется следующим образом.

В программе для открытия файла в языке C выполняется функция fopen.

FILE *fp=fopen(const char* file_name, const char* mode);

Тип FILE задает структуру блока управления файлом FCB (file control block). Память под этот блок отводится в области памяти процесса и он содержит текущую информации об открытом файле. Путь к файлу file_name позволяет найти функции fopen нужный индексный дескриптор файла в индексном файле.

В системной области памяти строится список дескрипторов открытых файлов. Каждый такой дескриптор содержит информацию из индексного дескриптора и два дополнительных поля: счетчик числа процессов, которые открыли этот файл совместно и тип открытия файла, который говорит открыт ли файл на чтение, на запись или как pipe. Этот тип файла означает, что pipe открыт одним процессом на чтение, а другим на запись.

Функция fopen просматривает список дескрипторов открытых файлов. Если требуемый файл находится в этом списке и открыт на чтение, то счетчик увеличивается на 1 и информация о первом блоке помещается в FCB.

Если дескриптор файла отсутствует в списке, то новый дескриптор добавляется к списку, а информация из индексного дескриптора файла переписывается в него. Соответственно пополняется информация и в FCB, куда заносится адрес дескриптора и устанавливается номер первого блока. В дальнейшем функции чтения или записи используют эту информацию и изменяют ее.

Разумеется, при анализе дескриптора файла проверяются полномочия доступа. Если они удовлетворяются, то происходит заполнение блока FCB и функция возвращает в указателе адрес на него. Этот указатель используется в функциях чтения и записи. Эти функции обращаются к системе ввода/вывода и запускают драйвер для чтения или записи очередного блока, выполняют блокирование и разблокирование записей и предоставляют логические записи программе и изменяют указатель на адрес следующего блока.

Освобождение информационного ресурса состоит в выполнении функции fclose, которая вычитает 1 из счетчика открытий файла. Если значение счетчика 0, то файл больше не используется и дескриптор файла удаляется из списка, а индексный дескриптор файла возможно модифицируется в индексном файле, восстанавливается первоначальная информация в FCB, а также очищается кэш от блоков файла и другой информации, связанной с файлом.

11.4 Характеристики файловых систем

Файловые системы различаются скоростью доступа, надёжностью хранения данных, степенью устойчивости при сбоях. Современные операционные системы поддерживают по несколько типов файловых систем. Помимо файловых систем, используемых для хранения данных на жёстком диске, система использует файловые системы CD/DVD, флэш устройств. Распространенным способом предохранить файловую систему от разрушения при сбоях является журналирование. Если система поддерживает ведение журнала, то есть ведет учет всех изменений на диске, то она используя эту информацию восстанавливается после сбоя. Надежность хранения данных обеспечивается методами репликации данных. Эти методы позволяют синхронно копировать данные при записи. Эти копии хранятся на разных носителях, возможно достаточно удаленных друг от друга.

Файловые системы Ext2/3 разработаны специально для Linux и традиционно используются на большинстве Linux-систем. Ext3 отличается от Ext2 только поддержкой журналирования, в остальном они одинаковы и легко могут быть преобразованы одна в другую в любой момент без потери данных. Обычно предпочтителен вариант с журналированием (Ext3) в силу его

большей надёжности. При высокой параллельной дисковой загрузке производительность Ext3 снижается, что выражается в снижении скорости операций с диском и повышении значения нагрузки на систему.

Файловая система ReiserFS похожа скорее на базу данных: внутри неё используется своя собственная система индексации и быстрого поиска данных, а представление в виде файлов и каталогов — только одна из возможностей использования такой файловой системы. Традиционно считается, что ReiserFS отлично подходит для хранения огромного числа маленьких файлов. Поддерживает журналирование.

Файловая система XFS, наиболее подходит для хранения очень больших файлов, в которых постоянно что-нибудь дописывается или изменяется. Поддерживает журналирование. Лишена недостатков Ext3 по производительности, но при её использовании выше риск потерять данные при сбоях питания (в том числе и по причине принудительного обнуления повреждённых блоков в целях безопасности; при этом метаданные файла обычно сохраняются и он выглядит как корректный). Рекомендуется использовать эту файловую систему с проверенным аппаратным обеспечением, подключенным к управляемому источнику бесперебойного питания (UPS).

Файловая система JFS Разработана IBM для файловых серверов с высокой нагрузкой: при разработке особый упор делался на производительность и надёжность, что и было достигнуто. Поддерживает журналирование.

Файловые системы FAT32/ FAT64 используются в разных версиях Windows, а также на многих съёмных носителях.

Файловая система NTFS изначально появилась в системах Windows NT, но может использоваться и другими версиями Windows.

ЗАКЛЮЧЕНИЕ

Эту книгу следует прочитать как минимум два раза. Дело в том, что автору приходилось использовать некоторые термины пояснение которых приходилось делать позже в соответствующей теме. Это связано со сложностью самого предмета изучения.

После освоения этого материала полезно изучить конкретные системы. Это можно сделать по специальной литературе, руководствам администратора и документации.

Список сокращений

ВС – вычислительная система
ИТ – информационные технологии
КС – компьютерная система
НД – набор данных
ОП – основная (оперативная) память
ОС – операционная система
ПО – программное обеспечение
СУБД – система управления базой данных
ЦП – центральный процессор
API – (Application program interface) программный интерфейс приложений
CAW (channel address word) – адресное слово канала содержит адрес канальной программы
CCW (channel command word) – команда канала
CSW (channel state word) – слово состояния канала
CHPID (channel path identifier)–идентификатор канального пути
DCB (data control block) – блок управления данными
DSCB (data set cSontrol block) – дескриптор НД
ECB (event control block) – блок управления событием
EXCP (execute channel program) – супервизор ввода/вывода
FCB (file control block) – блок управления файлом
IOB (input/output block) – блок ввода/вывода
IPL (initial program load) – программа начальной загрузки
JCL – язык управления заданиями (job control language)
JCT (job control table) – таблица управления заданием
JFCB (job file control block) – блок управления файлом задания
PAV (parallel access volume) – том с параллельным доступом
PCB – блок управления процесса
PID – идентификатор процесса
SCT (step control table) – таблица управления шагом задания
SIOT (step input/output table) – таблица ввода/вывода шага задания
SQA (system queue area) – область системных очередей
TIOT (task input/output table) – таблица ввода/вывода задачи
VTOC (volume table of content) – таблица содержимого тома
UCB (unit control block) – блок управления устройством
UIC (user identification code) – код идентификации пользователя

Литература

1. Мэдник С., Донован Дж. Операционные системы: Пер. с англ. М. : Мир, 1987. 792с.
2. Шоу А. Логическое проектирование операционных систем: Пер. с англ. М.: Мир, 1982
3. Дейтел Г. Введение в операционные системы: Пер. с англ. М.: Мир, 1987. т.1. 359с., т.2. 398с.
4. Таненбаум Э. Современные операционные системы (2-е издание) Изд-во «Питер» 2002г
5. Вильям Столлингс Операционные системы. 4-е издание 848 стр., с ил.;ISBN 5-8459-0310-6; формат 70x100/16; 2002, Изд. дом «Вильямс»
6. Э. Таненбаум, А.Вудхал. Операционные системы: разработка и реализация. 576стр. Изд-во «Питер» 2005г
7. Харрис Т., Бэкон Дж Операционные системы 1-е, 2004 800стр.
8. М. Руссинович, Д. Соломон М. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP, Windows 2000. Мастер-класс Издательства: Питер, Русская Редакция, 2005 г.
9. Губкин А.Ф. Операционные системы: Учебное пособие/ГЭТУ. – СПб.,1996. – 63с.
10. Олифер В.Г., Олифер Н.А. Сетевые операционные системы: Учебник для вузов. Изд.1-е, 2006 год 544 стр.
11. А.В.Гордеев. Операционные системы. 2 изд. 416стр. Изд-во «Питер» 2005г
12. Майк Эбберс, Уэйн О'Брайен, Билл Огден Введение в современные мэйнфреймы: основы z/OS. Москва 2007.
13. http://citforum.ru/operating_systems/

